Umeå University
Department of Computing Science
Master's Thesis
May 27, 2004

Accounting in Grid Environments

An Architecture Proposal and a Prototype Implementation

Author:
Peter Gardfjäll

Supervisor:
Erik Elmroth
Examiner:
Per Lindström

Abstract

This thesis proposes an accounting system architecture suitable for Grid environments. Furthermore, the design and implementation of a proof-of-concept accounting system prototype, based on the proposed architecture, is presented. The prototype has been developed to satisfy the accounting needs of SweGrid, a Swedish national Grid. The accounting system leverages state-of-the-art Grid and Web Services technology and offers transparency to Grid users, while providing fine-grained end-to-end security. Furthermore, the accounting system is based on open Grid standardization efforts and provides a single point of integration with underlying Grid middleware, allowing non-intrusive deployment in different Grid environments.

Initially driven by a need to tackle "larger-than-supercomputer" problems and enabled by improvements in networking technology, a new computing paradigm has emerged where geographically distributed computational resources can be connected to form a networked virtual supercomputer - popularly referred to as a *Grid*. Grid technology promises resource sharing on a global scale. Whereas the Internet provides uniform access to HTML documents, the Grid provides its users with on-demand access to all kinds of computational resources in a dynamic, distributed environment. A Grid allows a dynamic collection of individuals and institutions, referred to as a *Virtual Organization*, to share computational resources in a flexible, secure and coordinated manner.

The thesis investigates the area of *Grid accounting*, addressing challenges related to the problem of maintaining a Grid-wide view of Grid users' resource utilization. Accounting information can form a basis for evaluation and tracking of resource usage, Grid-wide enforcement of resource allocations and economic compensation. Collection and management of accounting information is important in any Grid environment, but in particular it is important to non-research organizations, which are not likely to go "on the Grid" unless they are paid for providing their resources.

Contents

1 Thesis Scope

This thesis proposes an accounting system architecture suitable for Grid environments. Furthermore, a prototype implementation based on the outlined architecture is presented. Whereas the architecture is presented in a general Grid setting, the prototype has been developed to satisfy the particular accounting needs of SweGrid, a Swedish national Grid.

The thesis project has been performed at the department of Computing Science at Umeå University under the supervision of Erik Elmroth. The project is part of a joint effort carried out in collaboration with Thomas Sandholm and Olle Mulmo at KTH, Stockholm.

The report is organized as follows. Section 2 gives a general introduction to Grid computing. It covers the background and motivation for Grids, as well as a definition of the Grid problem. Section 3 defines the Grid accounting problem and outlines the accounting needs of SweGrid. Section 4 introduces Grid standardization efforts and Grid technology. The accounting system architecture proposal, including design considerations and issues, is presented in Section 5. The prototype implementation is covered in Section 6. Section 7 discusses related work and Section 8 outlines topics subject to further investigation. Finally, Section 9 concludes the report.

2 The Grid: the Emergence of Virtual Organizations

2.1 The Benefits of Collaboration and Resource Pooling

Computing, both in science and industry, is increasingly concerned with collaboration and sharing of distributed computational resources, both within and across organization boundaries [13]. Collaborations can benefit from pooling of computational resources in several ways, e.g. by [11]:

- Utilizing a shared pool of high performance computers, storage and networks to tackle problems that are too large to be solved on a single system. This is sometimes referred to as *distributed super-computing*.
- Using otherwise unused processor cycles from idle computers in a shared pool of distributed workstations to increase aggregate throughput, much like the SETI@home project [32].
- Allowing on-demand access to remote resources, such as scientific instruments, which cannot be located locally for cost or practical reasons.
- Making proprietary software or database resources available to – and allowing greater sharing of results among – collaboration partners.

Today, collaboration is everyday practice in science and industry. High energy physicists from scientific centers all over the world pool computational, storage and network resources to perform analyses on the Petabytes of data that will be

generated by the Large Hadron Collider (LHC) [4] – a giant particle accelerator. Earthquake researchers spread all over the U.S. access remote scientific instruments and sensors connected to a linked set of computing resources to plan, perform and publish earthquake experiments [23]. In industry, organizations are increasingly cutting costs by outsourcing parts of their IT environments to service providers who offer continuous, on-demand access to resources [12, 13].

Generally, there seems to exist a need to engage in collaborative processes between geographically distributed groups of people. Collaborations, such as those described above, can vary widely in duration, size, structure, etc, but they all have a common need to share computational resources (be it data, storage, instruments, computing power, etc) in a controlled manner. It is exactly the class of problems encountered in such sharing arrangements, that is addressed by Grid technology.

2.2 The Grid Problem

Foster et al. [15] defines the "Grid problem" as being *flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions and resources*. I.e., Grids enable the sharing and coordinated use of resources in dynamic collaborations, allowing participants to make use of a common resource pool to tackle problems. Furthermore, sharing relationships that cross organizational boundaries, and hence security domains, may be established. A number of challenges arise in such sharing arrangements related to issues of security, resource discovery and resource access to mention a few. This class of problems is addressed by Grid technology.

Resource sharing in a Grid is not limited to file exchange. Rather, Grids provide on-demand access to all kinds of computational resources ranging from computers, storage and networks to data, software and scientific instruments. Note that resources are logical, rather than physical, entities. Single computers, entire computer clusters and distributed file systems all qualify as Grid resources. The sharing of resources is highly controlled. Resource providers and resource consumers need to negotiate resource sharing arrangements, defining the conditions of sharing, such as what is shared and who is allowed to access the shared resources. These sharing relationships may vary dynamically over time, e.g., in terms of the number of participants, what resources are shared and the access *policies* applied to the shared resources. A set of individuals and/or institutions participating in such sharing relationships are referred to as a *Virtual Organization* (VO). The collaborations presented in Section 2.1 are all examples of Virtual Organizations (VOs).

Sharing relationships are not limited to the client-server model. In fact, often sharing relationships are peer-to-peer; i.e., resource providers are resource consumers as well. E.g., a computation started on one resource may start a sub-computation on another resource.

From a user perspective, a Grid can be viewed as a *virtual computer*. I.e., Grid users have access to a virtual computing system, incorporating many diverse computing resources. The individual computing resources can be made invisible to the users, allowing users to consider the Grid as a whole [3].

To sum things up, Grids enable groups of people to form VOs and share computational resources in a controlled fashion, which allow members of the VO to collaborate in order to solve some common problem or reach a common goal.

2.3 Background: the Development of Meta-computers

The early development of Grid technology was motivated by scientific resource sharing applications, such as the pooling of distributed computational and storage resources for analyzing large data sets [12]. Driven by the need to solve problems which could not be solved by a single computer and propelled by technology improvements, particularly in network performance, the idea was to build virtual computers by connecting several distributed supercomputers, databases and scientific instruments, using high-speed networks. At that time the field was termed *meta-computing* and concerned with building *meta-computers*, i.e. networked virtual supercomputers [17].

The term *Grid computing* was coined in the mid-90s and origins from an analogy to the power grid. In the same manner people plug electrical appliances into receptacles to receive electrical power, Grid users should be able to "plug their applications into" the Grid to receive computational power. Much like in the power grid, Grid users should not need to be aware of the power source [3].

To date, mostly scientific applications have benefited from Grid technology. However, much like the Internet started out in a scientific environment and later reached widespread use in industry, Grids are expected to follow a similar path [13].

2.4 A Birds-eye View on Job Submissions in Computational Grids

This section outlines the entities and flow of events involved in a typical job submission scenario in a computational Grid. A *Computational Grid* focuses on aggregating compute power, e.g. by connecting high-performance computer clusters. Users submit computationally intensive jobs which are processed by the Grid.

2.4.1 Component Interactions During a Job Submission

Components of a Grid infrastructure, as well as their interactions during a job submission, are illustrated in Figure 1. Figure 1 is not meant to be prescriptive in terms of Grid infrastructural components. Rather, it illustrates components commonly employed in Grid environments.

1. A Grid user interacts with a user interface. The user interface which, e.g., could be a web portal or a simple command line interface, serves as an abstraction layer between the user and the Grid. It hides the complexity of the Grid from the user, allowing the user to view the Grid as a virtual computing resource. Through the user interface, applications are launched onto the Grid resources.

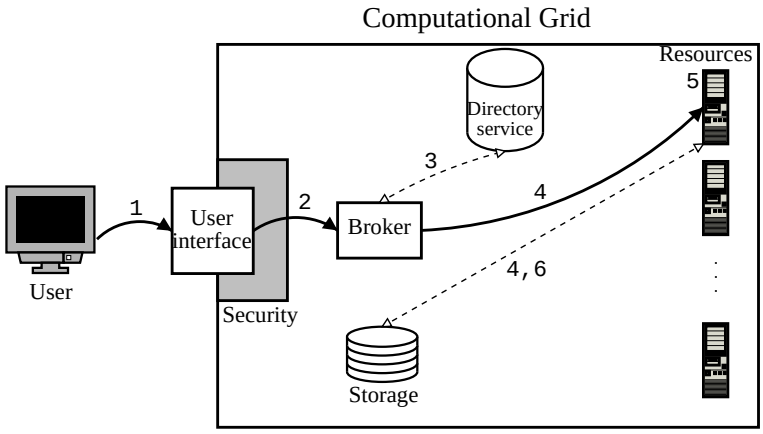

Figure 1: A typical job submission in a computational Grid.

2. Next, a resource broker, which might be integrated with the user interface, is contacted. The broker is responsible for identifying, characterizing and selecting appropriate resources for the job submission(s).
3. The broker contacts a directory service which provides dynamic information about available Grid resources. Based on the collected information and the user/application requirements an adequate (set of) resource(s) is selected. Resource requirements, such as approximated execution time and processor speed, is expressed through a job description language and is provided by the user/application as part of the job submission.
4. The broker submits the job(s) to the selected resource(s). The job executable as well as input data files are transferred to the resource(s), unless all data is already in place.
5. When executable and input data is in place, a resource starts processing its job (in case of a batch system the jobs might first have to wait in a job queue). Note that jobs can submit sub-jobs of their own to other resources.
6. At the time of job completion, any output files can be transferred back to the user or to another site for permanent storage. Output files are typically removed from the computing resources after some time.

2.4.2 Grid Security

The dynamic and cross-organizational nature of Grid sharing arrangements, makes security a major concern. To satisfy the strong security requirements of Grid environments, Grid technology makes heavy use of cryptographic techniques. In general, all communication between a user (or an entity acting on behalf of a user) and resources in a Grid is authenticated[1] and authorized[2]. Strong security mechanisms are crucial for enforcing sharing policies in a VO,

[1] Authentication is the process of determining someone's identity.
[2] Authorization is the process of determining someone's rights.

e.g. granting access to VO members only [41]. Figure 1 illustrates Grid security from a user perspective. Grid users authenticate themselves once and from that point on, all further security interactions are carried out transparently by Grid security mechanisms (see below).

The Grid Security Infrastructure (GSI) [41], developed by the Globus Alliance [36], provides security mechanisms commonly applied in Grid environments. In GSI, each Grid entity has a unique identity, be it a user, a resource or a process. The identity is represented by credentials, using an X.509 certificate. The certificate contains an entity name (subject), and a public-private key pair, and is signed by a Certification Authority (CA). The certificate subject is a globally unique Distinguished Name (DN), e.g., /O=Grid/OU=foo/CN=bar.

Two of the most important concepts related to Grid security are *single sign-on* and *credential delegation*:

- **Single sign-on**: a Grid user logs on to the Grid once (i.e., types a password). After that, the user will have access to all Grid resources where the user is authorized [41]. Re-authenticating (retyping password) on each resource interaction would be very inconvenient. In Figure 1, the user types the password once to the user interface. That creates a temporary time-limited proxy credential which acts as a stand-in for the user during the computation, eliminating the need for the user to retype his/her password on every security operation. Furthermore, the creation of a user proxy is a safety measure to avoid the (real) user credentials being compromised when sent across the network. The damage caused by a compromised user proxy is largely reduced by the fact that the user proxy has a bounded lifetime. Furthermore, a user can issue a restricted proxy with limited privileges to further reduce the potential damage caused by a compromised proxy [14].
- **Delegation**: since jobs can submit sub-jobs of their own and might need to access other resources, such as storage, a process must be able to act on behalf of the user. This is achieved through delegation of user credentials. In Figure 1, the broker delegates the user credentials to the resource on job submission and the resource runs the job using the delegated credentials, thus enabling the job to access all resources where the user is authorized. Delegation works recursively, in that a job can also delegate credentials to the sub-jobs it issues and so on. Each new proxy credential that is created is signed by its issuer, which can be the user or a process acting on behalf of the user. This establishes a chain of trust which can be traced all the way back to the CA which signed the initial certificate [14].

3 The Grid Accounting Problem

To date, Grid computing has mostly been confined to scientific environments. However, Grid computing is making its way into industry. Grid technology is now starting to gain industry support [10] and is reaching more widespread deployment in commercial settings.

However, as opposed to research organizations, most commercial organizations are not likely to go "on the Grid" unless they are paid for the resources they provide. In order to enable economic compensation, it is necessary to keep track of the resources utilized by Grid users. This is where Grid accounting enters the picture.

Here, we define Grid accounting as being *the process of maintaining a (consistent) Grid-wide view of VO members' resource utilization.*

The information acquired through Grid accounting can serve several useful purposes. E.g., it can:

- Form a basis for economic compensation. Once usage information is available, the resource provider could apply a transformation to convert resource usage into some monetary unit. Direct payments could also be envisioned, where resource usage is charged for immediately, e.g., by means of credit card transactions.
- Be used to enforce Grid-wide resource allocations. E.g., resources might only grant access to users whose current resource allocation has not been used up.
- Allow tracking of user jobs. Users can obtain information about a submitted job, such as where it was submitted, the resources it consumed and the output it generated.
- Enable evaluation of resource usage. Resource providers need to be able to determine to what extent different resources have been utilized. Furthermore, administrators can obtain information about what job executed on a specific resource at a certain time. Such information can be useful when debugging programs or tracking malicious users.
- Be used by resources to dynamically assign priorities to user requests based on previous resource usage.

Although great effort has been devoted to develop Grid standards, protocols and services, the area of Grid accounting has not yet been satisfactorily explored. A few other efforts have been targeting the Grid accounting problem, some of which will be discussed in Section 7.

3.1 SweGrid Accounting Specifics

SweGrid [35] is a computational Grid composed of six computer clusters with a total of six hundred CPUs. The cluster sites – located in Göteborg, Linköping, Lund, Stockholm, Umeå, and Uppsala – are connected through the GigaSunet high-performance network.

The computer time of SweGrid is controlled by the Swedish National Allocations Committee (SNAC) [33], which issues computer time, measured in *node hours* per month, to research projects. Grid-wide node hour allocations are assigned to projects based on their scientific merits and resource requirements. These node hour allocations specify the computer time per month that each project is entitled to. The whole allocation may be used up at one cluster or in parts

at any number of clusters. The accounting system must provide coordinated enforcement of project allocations across all resource sites.

A Grid accounting architecture is proposed in Section 5. A proof-of-concept implementation of the proposed architecture, for use in SweGrid, is presented in Section 6. But first, we introduce some key Grid technology and standardization efforts.

4 Grid Technology

The situation for Grids today resembles the pre-Internet situation for networks. I.e., Grids are currently not interconnected. Although many Grids have been built, the majority of them are not interoperable. In order to realize the vision of "*the* Grid" (c.f. Internet), it must be possible to establish sharing relationships between *any* potential VO participants, despite the use of different platforms, programming languages and environments. Interoperability is thus the primary concern. In networked environments interoperability means common protocols. Much like the Internet protocols and syntax (TCP/IP, HTTP, HTML) revolutionized networking by providing universal protocols for information sharing, standard *InterGrid protocols* are needed to allow universal resource sharing. Resources adhering to such an InterGrid protocol can be said to be *on the Grid* [10, 15].

To sum things up; for the long-term success of Grid computing there needs to be an agreement upon a common set of InterGrid protocols. Efforts in this direction are underway within the Global Grid Forum (GGF) [19], which is defining the Open Grid Services Architecture (OGSA), covered next.

4.1 Grid Standardization Efforts

OGSA merges efforts within the Grid and Web Services communities, to define the concept of a *Grid service*, a transient and stateful class of Web services, useful when building large-scale distributed systems (such as Grid systems). There are a few key concepts that the reader should be familiar with before turning our attention to the details of Grid services. These concepts are summarized in Figure 2.

- Building on concepts from both the Grid and Web Service communities OGSA [13] defines a Grid service; what it is, its capabilities, its behavior and how to interact with it. The definition is on a rather conceptual level. Furthermore, OGSA defines, scopes and outlines requirements for key services useful in Grid applications.
- Open Grid Services Infrastructure (OGSI) [16], on the other hand, is a formal and technical specification of the Grid service concept as outlined by OGSA, specifying fundamental behaviors and a core set of interfaces that defines a Grid service.

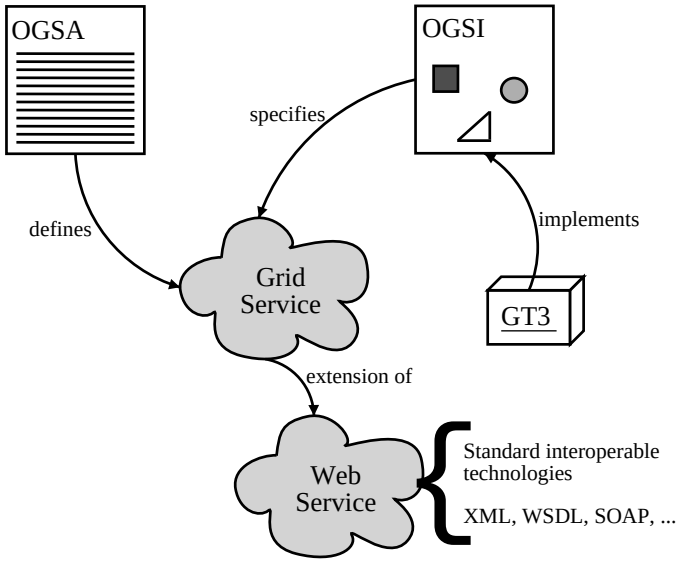

Figure 2: Key Grid service concepts and how they relate.

- Globus Toolkit 3 [31] is an open-source reference implementation of the OGSI specification, which allows programmers to implement and deploy their own Grid services.
- Grid services extend the capabilities of Web Services [38].

Next, these concepts will be presented in a little more detail. Web Services are introduced in Section 4.1.1 and Grid services are described in Section 4.1.2.

4.1.1 Web Services

Web services [38] represent a distributed computing technology, differing from other distributed technologies such as CORBA [28] and JavaRMI [22], by being based on the simple eXtensible Markup Language (XML) [9] standard. This property makes Web services independent of platform, operating system and programming language [13]. Web services are ideal in loosely coupled systems where interoperability is a primary concern, as opposed to most other contemporary distributed computing technologies, which are mostly used in highly coupled systems.

Web services include techniques for describing software components to be accessed and methods for discovering and accessing these components.

The capabilities offered by a Web service are defined in a service interface, or using Web service terminology – a *porttype*. Porttypes are specified using the Web Services Description Language (WSDL) [39], which can be described as an interface definition language for Web services. The service interfaces are defined abstractly, in terms of operations which exchange messages holding certain XML

payloads, effectively defining parameters and return values for the operations. These Web service interfaces are then bound to specific network protocols.

The Simple Object Access Protocol (SOAP) [34] represents one way of formatting remote invocations on Web services. SOAP in conjunction with HTTP is the most commonly used method to carry operation requests over the network. However, SOAP is only one way of formatting a Web service invocation. Other alternatives are conceivable as well.

4.1.2 Grid Services

In OGSA [13], all types of Grid resources (computers, storage, databases etc), are modeled as services – Grid services, to be more specific.

A *Grid service* is a *transient* and *stateful* Web service. I.e., Grid services can be characterized as a special class of Web services which maintain state between service invocations and have limited lifetime. A traditional Web service, on the other hand, is *stateless* (i.e., it does not maintain state between invocations) and *persistent* (i.e., it has an unlimited lifetime).

A particular instantiation of a Grid service, holding its own internal state, is referred to as a *Grid service instance*. For the purpose of distinguishing between different instantiations of a Grid service, each instance is assigned a globally unique identifier referred to as a *Grid Service Handle* (GSH). A GSH is a simple Uniform Resource Identifier (URI) such as http://host/services/MyGridService, and can be thought of as a permanent network pointer to a particular Grid service instance. Being a simple URI, a GSH does not carry enough information to allow a client to communicate with the service instance. Before communication can take place the GSH must be resolved to obtain a *Grid Service Reference* (GSR), which carries protocol binding information.

A Grid service instance is transient and as such it can be terminated. A service instance can either be destroyed explicitly or via a soft-state mechanism. To enable the latter, each Grid service instance is created with an initial *termination time*. When the termination time has expired the Grid service instance is terminated and any allocated resources can be released. Grid service instances expose operations both for explicit destruction as well as for updating the termination time. Soft-state protocols are particularly useful in distributed environments since they avoid the use of explicit "release" messages which can be lost in the network, potentially causing allocated resources to be "locked up" indefinitely. Soft-state protocols allow state to be discarded eventually unless "keep-alive" messages are received.

The transient nature of Grid services makes them suitable for representing not only physical resources but also more lightweight entities/activities, such as a video conferencing session, a data transfer or even a single database query.

All Grid services provide *service data*, which is a potentially dynamic set of *service data elements* (SDEs), holding XML-encapsulated information about the Grid service instance. Information such as metadata and internal service state is exposed through SDEs. All Grid services provide operations for querying and updating service data.

OGSI [16] defines a core set of composable interfaces which are used for constructing Grid services. The interfaces are defined in terms of WSDL portTypes, specifying the operations and service data associated with each service. The interfaces defined in OGSI are:

- **GridService**: defines basic behaviors common to all Grid services. This is the only interface that *must* be implemented by all Grid services. In fact, all portTypes below, except NotificationSink, extend the GridService portType.
- **Factory**: a Grid service that creates new Grid service instances.
- **NotificationSource**, **NotificationSink**, **NotificationSubscription**: interfaces that allows clients (NotificationSinks) to receive notifications on certain events in a NotificationSource. A subscription between a NotificationSource and a NotificationSink pair is represented by a NotificationSubscription service.
- **ServiceGroup**, **ServiceGroupRegistration**, **ServiceGroupEntry**: interfaces allowing the grouping of Grid services into ServiceGroups for such purposes as creating service indexes.
- **HandleResolver**: an interface for resolving GSHs into GSRs.

Table 1 presents an overview of two core interfaces, GridService and Factory, which the reader needs to be familiar with.

Table 1: The operations exposed by the GridService and Factory interfaces.

portType	**Operation**	**Description**
GridService	findServiceData	Query the service data of the Grid service instance.
	setServiceData	Update the service data of the Grid service instance.
	requestTerminationBefore requestTerminationAfter	Set the termination time of the Grid service instance.
	destroy	Terminate the Grid service instance.
Factory	createService	Create a new Grid service instance.

A portType does not have to define any operations; it might only define a set of SDEs. E.g., the ServiceGroupEntry portType does not define any operations of its own. It is merely a Grid service used to represent a member of a ServiceGroup and as such the lifetime management operations inherited from the GridService interface is enough. I.e., invoking the destroy operation on a ServiceGroupEntry service should remove the member service from its ServiceGroup.

A Grid service is constructed by combining a subset of the OGSI interfaces and any application specific interfaces defined by the developer. Figure 3 presents an abstract view of a Grid service implementing a set of interfaces, each with an associated set of SDEs.

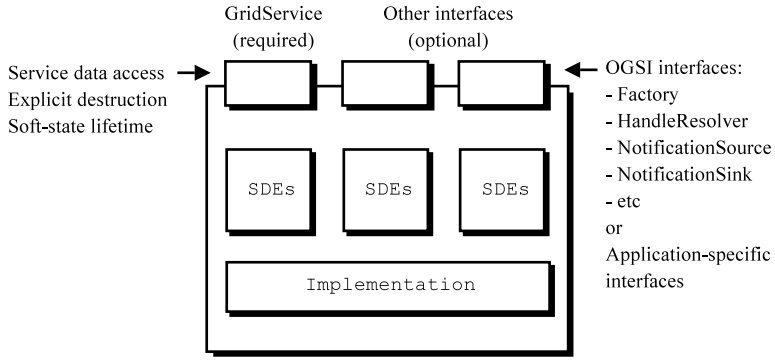

Figure 3: An abstract view of a Grid service.

Grid services can be composed to form higher-level services. However, the core set of interfaces defined by OGSI, which all Grid service implementations are based upon, allow us to treat higher order services uniformly no matter what they represent. E.g., since all Grid services share the basic behavior of the GridService interface, metadata about the service can be consistently obtained through the findServiceData operation, irrespective of whether the service instance represents a database query or a scientific instrument.

Figure 4 illustrates basic interactions between clients and service instances. A client creates a new Grid service instance by invoking the createService operation of a Factory (client 1) and a Grid service instance is terminated by invoking its destroy operation (client 4). During the lifetime of a service clients invoke the operations exposed by the interfaces implemented by the Grid service instance. Furthermore, several clients can invoke operations on the same service instance (client 2 and client 3).

4.2 Globus Toolkit 3 (GT3)

Globus Toolkit 3 (GT3) [31] is an open source reference implementation of the OGSI specification. In addition to the core interfaces specified by OGSI, GT3 also provides some useful services for resource management, data transfers and information services. GT3 enables developers to write Grid applications, by providing tools for implementing and deploying custom Grid services. The tools include a service container hosting Grid services and a WSDL-to-Java stub generator. GT3 is implemented in Java, and is based on the Axis SOAP engine [40].

GT3 represents a drastic paradigm shift compared to older toolkit versions. The previous toolkit versions were not based on OGSA, but provided services based on a rather heterogenous set of protocols. GT3 refactors all services in Globus Toolkit 2 (GT2) [17] and exposes them through the common abstraction of a Grid service [12].

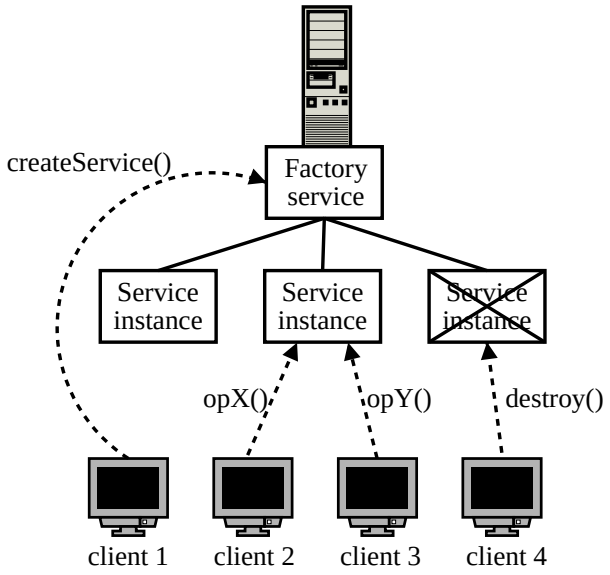

Figure 4: Basic client interactions with factory and service instances.

5 A Grid Accounting Architecture Proposal

In this section, an accounting system architecture, addressing the Grid accounting problem outlined in Section 3, is proposed. A more detailed architecture presentation can be found in [8]. The proposed architecture forms the foundation for the *SweGrid Accounting System* (SGAS) [7] proof-of-concept implementation presented in Section 6.

Two slightly different types of accounting models can be distinguished. We refer to them as *allocation based* and *aggregation based* accounting. In allocation based accounting each project is assigned a resource allocation (a quota) which may not be exceeded, unless a less strict quota enforcement policy has been specified. Hence, in allocation based accounting, quota enforcements are performed on service requests (job submissions[3]), possibly denying resource access if the allocation has been used up. To enforce Grid-wide quotas with strict consistency guarantees, resources must share a common view of each project's resource utilization. In order to maintain a consistent view of accounting information across the Grid, the information provided to resources must always be up-to-date.

Aggregation based accounting, on the other hand, does not enforce project quotas. Rather, usage information is aggregated as resources are used by a project. The collected information can, e.g., be used to determine a cost in real money at a later time. This means that unless strict timing guarantees are imposed on the accounting system (e.g. in terms of how rapidly accounting information needs to be available) an aggregation based approach can cache information locally

[3]In most practical cases, service requests are performed as job submissions. Hence, these terms will be used interchangeably although the former is more general than the latter.

at the resources, only updating global (i.e., Grid-wide) accounting information periodically, thereby increasing performance and scalability.

Aggregation based accounting can be viewed as a special case of allocation based accounting, in which the allocation is infinitely large (and hence no quota enforcement is necessary). An allocation based accounting model is assumed, since it represents a more general accounting model which is also in agreement with the needs of SweGrid.

The following assumptions are made about the Grid environment:

- a user can be a member of several VOs.
- a user can participate in one or more projects within each VO.
- each project has an account, with an associated resource allocation (i.e., a quota).
- accounts are handled by entities referred to as *banks*.

Before outlining the proposed architecture, the requirements and underlying design rationale of the architecture will be presented.

5.1 Functional Requirements

In any Grid environment, users request the services provided by resources[4]. This makes users and resources a good starting point for making a minimal list of required accounting system functionality.

From an accounting perspective, a resource must be able to:

- Provide users with *usage rates*, i.e., the cost associated with using a particular resource.
- Grant/deny users access based on their previous resource utilization, e.g., denying service requests by users having exceeded their allocations.
- Track resource usage.
- Charge project accounts for resource consumption.
- Provide resource usage information for each request that has been serviced.

A user, or an entity acting on behalf of the user, must be able to:

- Specify which project account should be charged for resource utilization.
- Obtain the usage rates associated with a particular resource.
- View the currently available quota (the account balance).
- Collect usage information from previous service interactions.

[4]Note that in the current SweGrid context, a resource is equivalent to a cluster, although other resource types might be considered in the future.

5.2 General Requirements

The large-scale, heterogenous, decentralized and distributed nature of Grid environments places special requirements on the accounting system. This section outlines some of the most critical requirements and how they are addressed by the proposed architecture.

5.2.1 Scalability

The accounting system must scale with an increasing number of users and resources. This out-rules a centralized solution where all Grid user accounts are handled by the same entity, as such an architecture would not scale. The proposed architecture suggests a decentralized approach in which each VO has its own associated bank which handles the accounts of the VO users.

Another scalability issue is Grid user to local site user mapping. To execute jobs on a resource, a Grid user needs a local user account on that resource. Current practice is for Grid users to be mapped to local accounts either using a 1:1 or an n:1 mapping. The 1:1 mapping (each Grid user is mapped to a separate local account) does not scale with many users and the n:1 approach (all Grid users are mapped to the same local account) might make it hard to track the resource usage of individual users and users might interfere with each other, e.g., overwriting each others files. Also, with an n:1 mapping all users have the same access rights, hence violating basic privacy requirements. A more flexible approach for user mapping is needed. An interesting alternative is based on each resource providing a pool of *template accounts*, which are temporarily linked to authorized Grid users for the duration of a job submission [1].

SGAS: The initial SGAS prototype does not incorporate template accounts.

5.2.2 User Transparency

From a usability perspective, an ideal accounting system is completely transparent to users. I.e., the end user should neither notice the presence of the accounting system, nor need to be aware of its existence. This is unrealistic in most cases, since a user participating in multiple projects will at least need to specify which project should be charged for the utilized resources. Transparency should also be offered to client-side programmers. The client-side should be kept clean from accounting system details. I.e., an application written without Grid accounting in mind should still be executable in an accounting-enabled system. Making the client-side transparent means placing more responsibility and trust on the resource side. The architecture provides user transparency by allowing resources to act on behalf of users.

5.2.3 Consistency

In order to provide strict enforcement of project quotas, a consistent view of accounting information must be maintained across the entire system. E.g., a

user with 1000 "credits" left on his/her account should not be able to submit six 1000-credit jobs concurrently to different resources and have them all run. The proposed architecture will provide consistency during normal circumstances. In case of failure of some critical entity, less strict levels of consistency can (optionally) be accepted.

5.2.4 Fault Tolerance

An accounting system should run smoothly even in the event of component failures. In order to enforce allocations, it is critical that the accounting system exhibits a high level of availability. Hence, in order to provide high availability and fault tolerance, some of the key accounting system components might need to be replicated. However, in a system with strict consistency requirements, synchronization between replicas might introduce significant performance overhead. Moreover, a non-replicated system can still be distributed, reducing the impact of single component failures. E.g., a bank could be implemented as a set of distributed "sub-banks", holding non-overlapping subsets of project accounts.

Failures occur even in replicated systems. Despite a highly available system, some soft failure handling mechanism is needed in case disaster strikes. In case of an unavailable accounting system, it is not acceptable to deny all service requests made by VO members. In such cases users should (optionally) be granted access to resources but with less strict consistency guarantees.

SGAS: The architecture neither prescribes nor prevents the use of replication. The first prototype of SGAS does not incorporate replication, although it might be introduced later. The prototype system will try to mask failures by proceeding as if nothing is wrong, caching accounting information locally on the resources and synchronizing at a later time when the failed component is up and running again. During such periods of downtime, strict enforcement of quotas might not be possible. Users worried about exceeding their allocations can (optionally) specify that their requests should only be serviced if contact with the accounting system can be established.

5.2.5 Trust and Security

Accounting information must only be exchanged between trusted entities. Thus, the accounting system needs to maintain some notion of trusted users (e.g., project members) and privileged users (e.g., system administrators) and their associated access rights. Access will only be granted to entities capable of authenticating themselves as a trusted or privileged user. The architecture relies on the Grid security mechanisms outlined in Section 2.4.2 to provide end-to-end security.

SGAS: The initial SGAS prototype provides a fine-grained authorization framework (see Section 6.2.4) allowing individual access permissions to be dynamically associated with each project member. SGAS will assume pre-existing trust relationships among the entities involved in a service exchange. However, in Section 5.5.2 a more distrustful scenario is investigated and a potential solution is discussed.

5.2.6 Ease of Deployment

Grid environments are inherently heterogenous. Furthermore, no single standard Grid middleware solution currently exists. Most Grids built to date, are not interoperable and differ widely in structure and underlying software stacks.

To reach widespread deployment, the integration of the accounting system into different Grid environments must be straightforward, not requiring large modifications to the existing environment. Due to the lack of commonality between different Grid solutions, no generalizing assumptions should be made about the underlying environment. The accounting system must be modular, flexible and to a large extent developed independently of any particular Grid environment.

The proposed architecture offers light-weight deployment by providing a single-point-of-integration with underlying Grid environments. Furthermore, the accounting system is not restricted to a particular set of resource types, but is flexible enough to accommodate arbitrary types of resource usage, as long as the underlying environment is able to measure the consumption of each resource type.

5.3 Design Considerations

A wide range of design alternatives exist for the accounting system. This section outlines some of the design options and motivates the choices made for the architecture and SGAS prototype.

5.3.1 Type of Bank

Depending on the consistency and timing requirements of the accounting system, different bank approaches are conceivable.

In an *online bank* approach resources always examine the user account prior to servicing a request. Access decisions are based on the amount of available funds in the user account. If a user submits a job without sufficient funds available, the resource may not allow the job to run. This approach allows strict enforcement of project quotas.

An *offline bank* approach is also conceivable, where resources base their access decisions on locally cached accounting information. The resources periodically synchronize with a bank to obtain up-to-date accounting information. The offline bank approach might be useful in a VO with less restrictive accounting information consistency and timing requirements. In such cases, scalability can be achieved by employing a hierarchy of (offline) banking services. The drawback of this approach is that it would be hard to use in an allocation based accounting system where quotas are strictly enforced.

A *gather/scatter* approach represents another solution where allocations are spread across resources and resource usage information is gathered from all resources when needed. The gather/scatter approach has the apparent drawback that an allocation cannot be spent arbitrarily among the resources.

SGAS: The strict allocation enforcement requirements of SweGrid makes the online bank approach the only viable option. SGAS will probably allow for some temporary exceeding of funds, at least in an initial deployment phase. However, the handling of allocation exceeding is a policy issue rather than an architectural one. The architecture must not stipulate a particular policy which is applied in every situation. Rather, the architecture should provide sufficient support in order to accommodate and enforce any choice of policy. Policy decisions are left to users, resource managers and allocation authorities.

5.3.2 Types of Resource Usage

The types of resource usage being accounted for depends on what resource usage information can be collected by resources as well as the individual needs of resource providers. All resource providers will not charge their users for all types of resources. Some resource types which might be accounted for are:

- CPU-time.
- Wall-clock time.
- Number of processors.
- Storage.
- Network bandwidth.
- Data transfers.
- Memory.
- Quality of Service (e.g. higher batch queue priority).
- Software usage.

SGAS: In SweGrid only wall-clock time, i.e. node-hours, will be considered initially, but as SweGrid evolves other resource types, such as storage, might be accounted for as well. SGAS is designed with flexibility in mind and does not impose any restrictions on the types of resource usage accounted for.

5.3.3 Currency

Accounting for different resource types raises the question of what "currency" should be used to charge for resource usage. At least two approaches are conceivable:

- Use separate currencies for different resource types, i.e., maintain a separate quota for each type of resource.
- Convert different types of resource usage into a single currency unit. Such an approach generally requires some function to map resource usage into that currency.

SGAS: The accounting system employs a unit-less currency called *Grid credits* which can be used to charge for arbitrary resource usage. Since SweGrid users are only charged for node-hours, the two approaches are equivalent. Thus, for

an initial SweGrid setting, no function is needed to convert resource usage to Grid credits. If additional resource types need to be accounted for a mapping function will need to be provided, unless resource allocations are dedicated for each resource type, in which case a separate account can be setup for each type of resource. Thus, the accounting system also supports the first approach.

5.3.4 Resource Valuation

Different resources have different performance characteristics. It could be argued that it should be less expensive to run a two hour job on a Pentium II processor than on a Pentium 4 processor, since during a two hour period more operations would be performed on the Pentium 4. The issue at hand is whether resources should be valued based on their characteristics, such as performance.

SGAS: SGAS resources (i.e. clusters) will initially be uniformly valued. However, for a general heterogenous Grid environment, such an approach would not be satisfactory. In the general case, resource providers should be allowed to publish the *usage rates* (i.e., the costs) of their resources, creating a kind of Grid market place where resource providers compete for clients. To this end, resource cost negotiation scenarios can be envisioned, where requesters (clients or services) negotiate usage rates with providers prior to resource usage.

5.3.5 Obtaining Usage Rates

In order to make an informed decision on the choice of resource(s), a user, or an entity acting on behalf of the user, must be able to obtain usage rates associated with each resource. A few approaches are conceivable:

- Uniform usage rates are used throughout the Grid.
- Clients query resources for their usage rates.
- Resources publish their usage rates through an information system.

SGAS: The usage rates will be uniform (e.g., one Grid credit per node-hour) in an initial SweGrid deployment. A long-term solution should incorporate at least one of the other alternatives.

5.3.6 Time of Payment

At least three alternatives exist, as to when a project account can be charged for resource utilization:

- *pre-payment*: pay before resource usage.
- *continuous-payment*: pay continuously as resources are utilized.
- *post-payment*: pay after resource usage.

The pre-payment approach is inherently difficult due to the problem of estimating an accurate runtime prior to job submission. The continuous-payment

approach might place a heavy load on the accounting system, making the scalability of that approach questionable.

SGAS: The last alternative seems to be the only viable option. If project quotas need to be enforced, this approach relies on an ability to reserve the necessary funds from the project account prior to resource utilization (c.f. making reservations on a credit card). A resource will only grant a client access if sufficient funds can be reserved from the client's project account.

5.3.7 Auditing

It is likely that resource providers, VOs, allocation authorities and users are interested in detailed information regarding job execution and resource usage. Such information could, e.g., be used to evaluate resource usage or to forecast future usage patterns. Moreover, some sort of *job tracking* functionality could prove very useful. I.e., users would not only have access to information about the resources consumed by their jobs, but also detailed information about the status of jobs (whether a job completed, failed, was canceled, etc), the time of execution, job input data, job output data, etc. Such information could also be used by resource administrators, e.g., to track a malicious user/job.

SGAS: The accounting system includes a *logging service*, through which detailed information about jobs and resource usage can be published and queried.

5.4 Architecture Overview

This section presents an overview of the proposed accounting system architecture. The accounting system entities as well as their interactions are outlined.

5.4.1 Entity Overview

Figure 5 illustrates the main entities involved in the accounting system architecture. The shaded components identify accounting system entities.

Each VO has an associated *Bank service* to manage the accounts of the VO projects. A one-bank-per-VO limit might seem restrictive in a general, large-scale Grid environment. One concern might be that a single bank will get scalability problems as the VO grows. However, it should be stressed that the architecture says nothing about the implementation of the bank. The bank is not confined to a single site. It could be implemented as a virtual resource, composed of several distributed services to achieve load-balancing and scalability. Another cause of concern is a scenario in which an allocation authority, such as SNAC, does not trust the existing VO bank and would like to deploy its own bank. In such case, the allocation authority should probably set up a new VO with its own bank, rather than deploying a second bank for the existing VO.

On each Grid resource, an *Accounting manager* (AM) handles incoming service requests, performs account reservations, and charges the requester's account after resource usage.

A *Log and Usage service* holds detailed information about submitted jobs, such as their resource consumption.

5.4.2 Entity Interactions

Interactions among the accounting system entities are exemplified in a job submission scenario where a job is submitted to a computer cluster, although it could be generalized to cover a generic service request to an arbitrary Grid resource. Figure 5 outlines the main interactions.

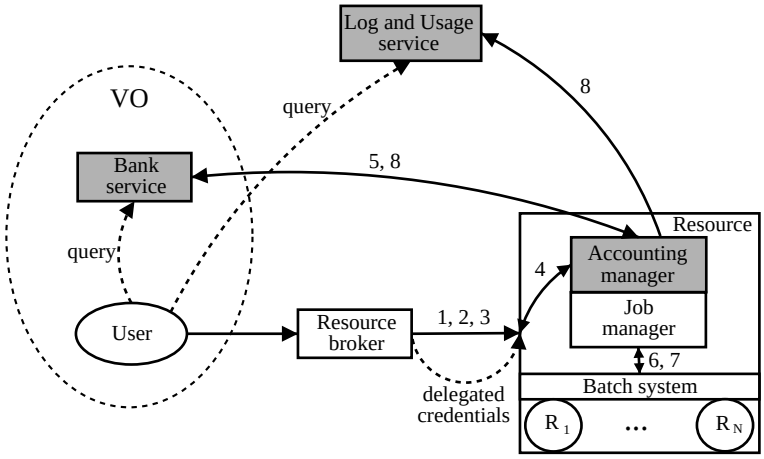

Figure 5: Interactions among accounting system entities (shaded) during a typical job submission.

1. The user, or an entity acting on behalf of the user, e.g. a broker, contacts the resource that has been selected to execute the job.
2. The user and the resource mutually authenticate each other and the resource authorizes the user. The user credentials are delegated to the resource, making the resource capable of acting on behalf of the user.
3. The user submits a job request to the resource. The job request includes the project account to be charged (and implicitly the bank service).
4. The AM, controlling a Job Manager (JM), intercepts the job request. The AM ensures that the specified VO bank is trusted before proceeding.
5. The AM acquires a time-limited (soft-state) hold on an amount of the project account. The hold acts as a reservation on a portion of the project allocation, guaranteeing the AM that sufficient funds will be available when the job finishes. The reserved amount is based on the run-time approximation in the provided job request. The AM authenticates with the bank using the delegated user credentials. The bank grants the hold if the user is recognized as a trusted user and sufficient funds are available in the project account. If the hold can be acquired, a hold-id is returned to the AM. The hold-id is used later by the AM to refer to the hold.

6. If the hold is granted, the AM forwards the job request to the JM, which submits the job to the local resource manager (e.g., a batch system). If necessary (e.g., in case of a long batch queue) the AM can extend the lifetime of the hold.
7. After the job has completed (or canceled) the AM gathers information about the resources consumed by the job and records this in a Usage Record (UR).
8. The AM charges the project account for the resource usage, by using the hold, and logs the UR in the Log and Usage service. Any residual amount of the hold is returned to the account.

The bank service as well as the Log and Usage service can be queried by users. E.g., a user might query the bank to get a list of his/her latest transactions. For detailed information about particular jobs, queries can be posed to the Log and Usage service. A bank transaction entry and a log entry are associated by a unique (Grid-wide) job identifier.

5.5 Architecture issues

There are a few issues associated with the proposed architecture, which need to be addressed by an implementation, or at least taken into consideration.

5.5.1 Soft-state Holds

As mentioned above, the reservation of user funds – the acquiring of a hold – prior to job submission must be performed using a soft-state approach. With hard-state holds, where explicit "release hold" operations are required, a resource crash after acquiring a hold could have the devastating effect of locking up a portion of an account indefinitely. A problem associated with the use of time-limited holds is that estimating the job runtime (including batch queue time) might be difficult. Hence, some means must exist to extend the lifetime of a hold after its acquiring.

5.5.2 Trust and Service-agreements

In a general accounting system, the user and the resource should come to a mutual agreement about the terms of the resource usage. This implies having both parties sign some form of contract or agreement. The bank only services resources providing a valid agreement. We refer to such a service agreement approach, where the bank requires a contract signed by both parties to grant the resource access to an account, as a *contract signing* approach.

The contract signing approach relies on the user only delegating restricted proxies (the default behavior), which cannot be used for impersonation.

Figure 6 shows interactions which could be involved in a service request using a rudimentary contract signing approach.

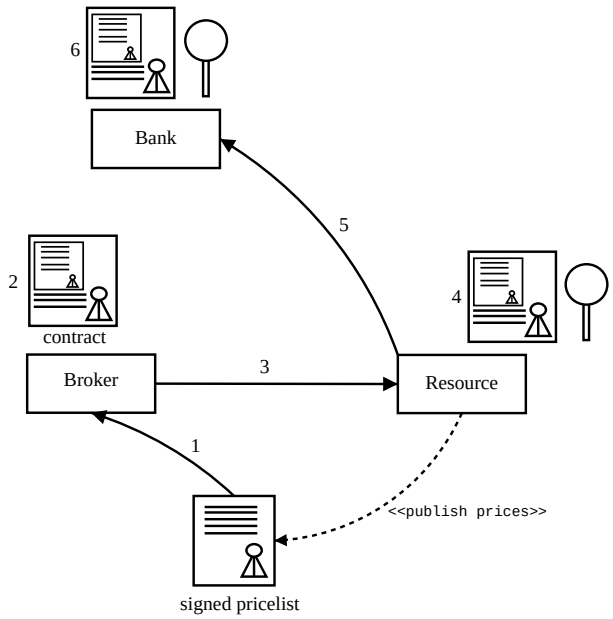

Figure 6: Interactions in a crude contract signing service agreement.

1. The user, or an entity acting on behalf of the user, receives a signed list of usage rates from the resource.
2. If the rates are acceptable, the user appends the approximated run time to the rate list and signs the entire document. This document constitutes a valid service agreement contract.
3. The contract is sent to the resource together with the job request.
4. The resource makes sure that the contract is valid and that the terms are acceptable. The resource then appends the contract to its request for a hold on the user account.
5. The hold request is sent to the bank.
6. The bank checks the contract and makes sure that both signatures are valid and performed by full proxies. If the hold can be acquired, the bank associates the contract with the hold.
7. When the job finishes, the resource sends a UR and the hold-id to the bank. The bank uses the resource usage information in the UR together with the prices stated in the contract to withdraw the correct amount from the account. Cost calculations could be performed by a trusted third-party service, since cost calculations are outside the scope of functionality which should be provided by a bank service.

The proposed architecture, as outlined in Figure 5, does not employ a contract signing approach. Rather, it assumes pre-existing trust relationships between the involved entities. Resources only authorize trusted users, and always confirms that the bank identity is well-known. Banks only grant trusted users access

to accounts. Since resources impersonate users, by using the users' delegated credentials, banks do not authorize the resources themselves. This means that the user must always submit jobs (and hence credentials) to trusted resources. Otherwise, nothing protects the user account from a fraudulent resource. An alternative approach which does not rely on the use of impersonation and delegated credentials is to have the bank keep a list of trusted resources, giving these resources full rights to update accounts. We refer to these kinds of service agreements, where a user blindly trusts the resource to act on his/her behalf, as *in-blanco signing* approaches. I.e., the user implicitly "signs" a contract giving the resource full access to the project account. Users thus need to trust resources to withdraw correct amounts from their accounts. The main motivation behind the in-blanco signing approach is the transparency it offers to users.

5.5.3 Co-allocation

The potential use of co-allocation needs to be taken into account. Specifically, a co-allocator might want to submit a number of jobs to different resources on behalf of a user. Moreover, these jobs should be run together, atomically, in the sense that either all jobs are successfully submitted or none is. It would not be acceptable to submit nine jobs successfully, while the tenth job submission fails, due to lack of user funds. Meanwhile, the nine jobs already started would be consuming resources while waiting for the tenth job, without getting any useful work done.

The responsibility of validating the availability of necessary funds for co-allocated jobs should not be placed on the user. Even if the user could validate the existence of sufficient funds, nothing guarantees that those funds would be available at the time of submission. E.g., another user of the project might have submitted a job meanwhile.

The accounting system needs to provide means to allow user transparent co-allocation. A general solution would be to have the co-allocator acquire holds for all (co-allocated) jobs prior to submission. These pre-acquired holds could then be handed over to the resources responsible for executing the jobs.

SGAS: The passing of pre-acquired holds to a resource is currently not supported by SGAS. However, it should be a rather straightforward extension. Furthermore, co-allocation is a feature that will not be used extensively in an initial SweGrid setting. In fact, no co-allocation service is currently available in the SweGrid environment.

6 The Design and Implementation of SGAS

This section covers the design and implementation of the SGAS prototype. SGAS is built around a set of OGSI-compliant Grid services. There are three main components of SGAS, each with a one-to-one correspondence to a component of the accounting system architecture illustrated in Figure 5.

The *Log and Usage Tracking Service* (LUTS) corresponds to the "Log and Usage Service" of Figure 5. Its main responsibility is to collect and publish URs, holding detailed information about the resources consumed by jobs. It enables publishing and querying of URs through the setServiceData and findServiceData operations defined by the common OGSI::GridService interface.

The *Job Account Reservation Manager* (JARM) corresponds to the "Accounting Manager" of the presented architecture. JARM serves as the (single) point of integration between SGAS and the underlying Grid environment. On each Grid resource, a JARM intercepts incoming service requests, performs account reservations prior to resource usage, and charges the requester's account after resource usage. The underlying Grid environment does not need to be OGSI-compliant. In fact, the first version of SGAS will be deployed on top of the NorduGrid [26] middleware which is based on GT2, and hence is not OGSI-compliant. JARM is not a Grid service – it is a client invoking operations on the other SGAS Grid services.

For more details on these two components, the reader is referred to their design documents [29, 30]. Prototype implementations of these two components have been developed by Thomas Sandholm, at KTH, and will not be discussed further in this report, which focuses on the Bank component of SGAS, presented next.

6.1 SGAS Bank Design

The design of the SGAS Bank [6] component is thoroughly presented in a separate design document [18]. The Bank component, which corresponds to the "Bank service" of Figure 5, is primarily responsible for maintaining a consistent view of the resources consumed by SweGrid projects.

The bank is designed with generality and flexibility in mind. Specifically, the bank does not assume any particular type of resources and as such it can be integrated into any Grid environment. E.g., as outlined in Section 5, all bank transactions are performed using Grid credits, a unit-less currency which can be used to charge for arbitrary resource usage. Resources map resource usage into Grid credits before charging an account.

The bank is also neutral with respect to policies. Policy decisions are left to clients, resources and bank administrators. The bank provides the flexibility to accommodate such policies, as well as means of enforcing them. E.g., policies dictated by the allocation authority or the resource might allow a job to run even though the project allocation has been used up. However, if project quotas are not strictly enforced by a resource, the user can still decide only to perform "safe" withdrawals that do not exceed the project allocation.

6.1.1 Bank Services

The SGAS Bank is composed of three tightly integrated Grid services (`Bank`, `BankAccount` and `AccountHold`). An additional service interface, `Service-AuthorizationManagement` (`SAM`), has been developed to support fine-grained

authorization of service users. The interfaces and their relationships are illustrated in Figure 7.

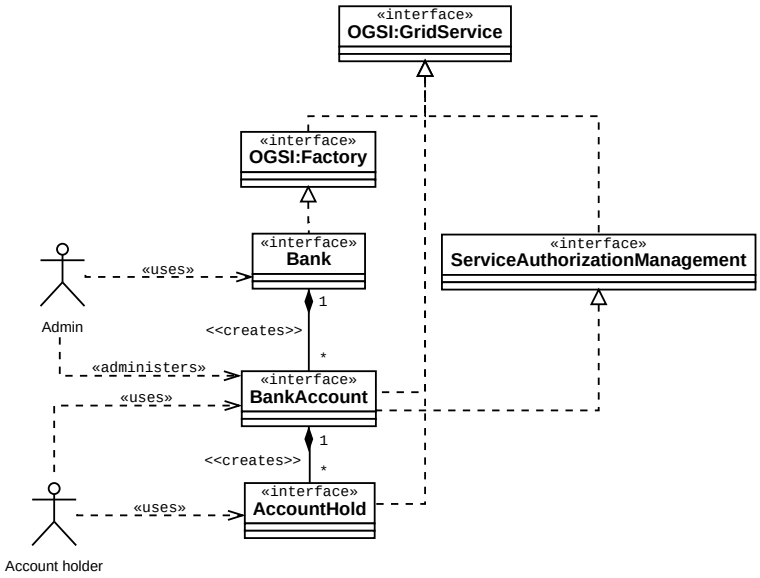

Figure 7: Bank interface relationships.

Bank service. The `Bank` service is responsible for creating as well as locating `BankAccount`s. The `Bank` service implements the factory pattern as provided by OGSI. This allows the `Bank` to create new `BankAccount` service instances. Clients can obtain a list of the `BankAccount`s they are authorized to use by querying the `Bank`.

BankAccount service. A `BankAccount` service manages the resource allocation of a research project. A project member can request a hold on the `BankAccount`, effectively reserving a portion of the project allocation for a bounded period of time. A successful hold request results in the creation of an `AccountHold` service, acting as a lock on the reserved account quota. We refer to the `BankAccount` service that created an `AccountHold` service as the *parent account* of that hold.

The set of authorized `BankAccount` members, as well as their individual access permissions, is dynamic and can be modified at run-time. To this end, the `BankAccount` interface extends `SAM`, a general-purpose service interface allowing a Grid service to maintain and modify a set of associated *service users*. Each service user has an associated access control list (ACL) specifying his/her permitted operations. E.g., `SAM` allows restricted ACLs to be associated with regular account holders, while administrators can enjoy full access permissions. Note that there is nothing bank-specific about `SAM`; it can be used by any service requiring fine-grained management of service authorization.

`BankAccount` services maintain and publish transaction history and account state, which can be queried by authorized `BankAccount` members.

AccountHold service. `AccountHold` services represent time-limited reserva-

tions on a portion of the parent account's allocation. `AccountHold`s are usually acquired prior to a service request and are committed (i.e., cashed in) after the request has been serviced to charge their parent account for the consumed resources. `AccountHold` services are created with an initial lifetime and can be destroyed either explicitly or through expired lifetime. `AccountHold` expiry is controlled using the lifetime management facilities provided by OGSI. On commit or destruction, the `AccountHold` service is destroyed and any residual amount is returned to the parent account. In the case of destruction, the entire `AccountHold` reservation is released. On commit, a specified portion of the `AccountHold`, corresponding to the actual resource usage, is withdrawn from the parent account, and a transaction entry is recorded in the transaction log.

6.1.2 Service Interfaces

The operations exposed by the banking Grid services are presented in Table 2. Note that the service interfaces extend OGSI interfaces, whose operations (Table 1) are used for such purposes as lifetime management, instance creation and service data access.

Table 2: The operations exposed by the SGAS Bank interfaces.

portType	**Operation**	**Description**
`SAM`	addServiceUsers	Adds a set of service users with specified access rights.
	updateServiceUsers	Updates the access rights for a set of service users.
	removeServiceUsers	Removes a set of service users.
`Bank`	getAccounts	Get GSHs of all accounts where caller is an account holder.
`BankAccount`	requestHold	Creates an account hold if enough funds are available.
	addAllocation	Adds a (potentially negative) amount to the account allocation.
`AccountHold`	commit	Withdraws a specified amount of the account hold from the parent account.

Table 3 gives an overview of the service data exposed by the banking Grid services. Note that the service data set of each service also contains the service data of extended OGSI interfaces. Furthermore, note that since `BankAccount` extends `SAM`, a `BankAccount` instance also exposes the serviceUsers SDE.

6.1.3 Bank Interactions

Figure 8 illustrates typical interactions between an administrator and the bank. All operations invoked in Figure 8 should typically require administrator privileges, preventing regular users from accessing them.

Table 3: The service data exposed by the SGAS Bank interfaces.

portType	**serviceData**	**Description**
`SAM`	serviceUsers	The set of service users currently associated with the service and their access rights.
`Bank`	none	-
`BankAccount`	accountData	The state of the account: total allocation, currently reserved and spent funds.
	transactionLog	Holds all transaction log entries and can be queried through XPath.
`AccountHold`	accountHoldData	The hold amount and the parent account GSH.

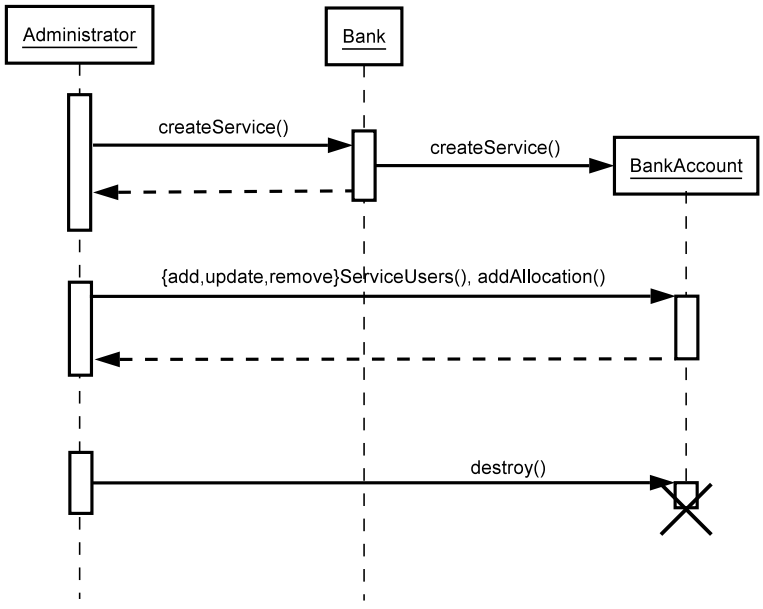

Figure 8: Administrator interactions with a `BankAccount` instance.

When a new project is initiated, a `BankAccount` instance, representing the initial project allocation, is created via the `Bank`. During the lifetime of a `BankAccount`, the set of account holders can be modified and the account allocation can be updated. The addServiceUsers, updateServiceUsers, removeServiceUsers and addAllocation operations are used for such purposes. `BankAccount` instances are explicitly destroyed when their corresponding research project has ended.

Figure 9 shows interactions between a resource and a bank at the time of job submission. The resource, running a JARM, acts on behalf of the user that submitted the job. Unless the user specifies a particular account in the job request, the resource calls the getAccounts operation on the `Bank` to get a list of the user's `BankAccounts`. Once a valid GSH for a `BankAccount` is obtained, a hold is created to make a reservation on the account allocation. The resource provides an initial termination time on hold creation. If the `AccountHold` is destroyed, either through an explicit call to the destroy operation or through expired termination time, the reservation is released. The termination time can be reset at any time using the requestTerminationAfter operation of the OGSI:GridService interface. The commit operation withdraws a specified amount of the hold, corresponding to the actual resource usage, from the parent account. Any residual amount of the hold is returned to the account. On commit, the `AccountHold` service instance is terminated.

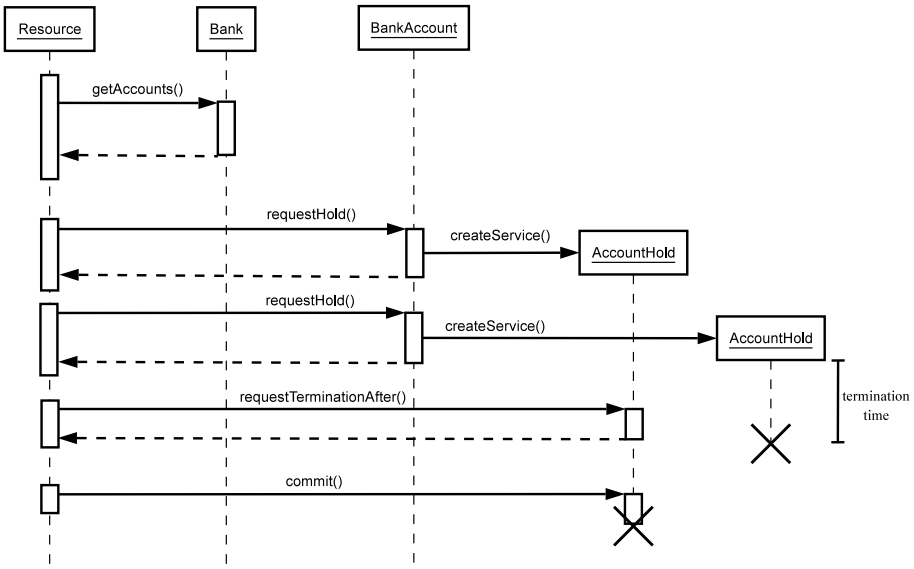

Figure 9: Resource interactions with the bank.

6.2 SGAS Bank Implementation

The SGAS Bank prototype implements the Grid services outlined in Section 6.1. All Grid services are implemented and deployed using the tools provided by GT3 [31].

6.2.1 Inheritance and Delegation

Basically, all banking services are implemented as Java classes using the GT3 framework. Implementation of a Grid service in GT3 can be performed according to one of two approaches:

- In the *inheritance approach* the Grid service class inherits the GridServiceImpl class – the GT3 implementation of the OGSI:GridService interface.
- Using a *delegation approach*, a set of operation providers, each implementing a disjoint set of operations, are combined to form the Grid service. Service invocations are then dispatched to the operation provider which implements the operation being invoked.

The delegation approach is more flexible since it allows generic interface implementations to be reused by other Grid services, and facilitates a development model based on composition of primitives. E.g., any service requiring fine-grained management of authorization (such as `BankAccount`) can include the `SAM` operation provider in its implementation. The other, less general, services are currently implemented using the inheritance approach.

6.2.2 Persistency – Xindice XML:DB

In order to guarantee recoverability, all service state is checkpointed to a database. The services make use of an XML database, called Xindice [2], to store their states. Xindice is an open-source database for storing XML data, which conforms to the standards developed by the XML:DB group [43]. Xindice can be accessed by Grid services in GT3, since a Xindice database driver is provided in the GT3 distribution.

In the event of a server crash the state of all services needs to be recovered from secondary storage on server restart. Besides enabling state checkpointing and recovery, the database solution further allows users to pose non-trivial queries against banking information, such as the transaction log, by exposing database content as service data, and embedding database query expressions in service data queries. The GT3 query evaluator framework allows service data queries to be redirected to a database client, effectively exposing a database view through the service data framework. E.g., a `BankAccount` member can pose XPath [42] queries against the transaction log to obtain specific transaction information. The XPath query approach allows users to find information by means of XPath queries, with well-known semantics, and it further avoids the inconvenience of defining a query language by means of different interface operations. XPath is a rather simple but powerful query language to retrieve a set of elements, subject to certain restrictions, from an XML document.

Prior to database storage the service state is serialized into an XML document. E.g., the state of a `BankAccount` instance is serialized into the following XML structure:

```
<bankAccount>
  <state>
    <project> .. </project>
```

```
    <funds>
       <allocation> .. </allocation>
       <spent> .. </spent>
       <reserved> .. </reserved>
    </funds>
  </state>
</bankAccount>
<transaction> .. </transaction>
    ...
<transaction> .. </transaction>
```

The serialized `BankAccount` state includes the name of the owning research project, the total allocation, the currently spent and reserved funds, as well as the transaction history of the account.

Account information can be queried by users through the findServiceData operation. E.g., a query embedding the XPath expression

```
/bankAccount/funds/*
```

would return the allocation, spent and reserved entries, whereas

```
/transaction[userDN='/O=Grid/OU=foo/CN=bar']
```

retrieves all entries in the `BankAccount`'s transaction log made by the Grid user with DN /O=Grid/OU=foo/CN=bar.

Xindice is missing an important piece of functionality, commonly applied in database settings – transactions. This limitation forces us to serialize database access to an account through a single entity – in this case the `BankAccount` instance itself. Unless we serialize updates to the database we might, and in the long run we *will*, experience *lost updates*[5] and other problems related to unsynchronized database access. This has implications for the implementation.

The most straightforward solution would have been to have the `AccountHold` update the database state of the parent account on termination (i.e., commit, destroy or lifetime expiry), but that approach is prevented by Xindice's lack of transaction support. Instead, the `AccountHold` will perform a callback to the parent `BankAccount` instance, which will perform the database update itself, ensuring that updates to the database are serialized. The use of callbacks and database calls is shown in Figure 10, which illustrates a scenario where two `AccountHold`s are created. One `AccountHold` is committed and the other `AccountHold` expires. Both events result in callbacks to the parent account of the `AccountHold`, which serializes database access.

The implementation would have been greatly simplified if Xindice had provided ACID[6] property transaction support. Furthermore, account database updates might have been more efficient, e.g., if a more fine-grained locking scheme was implemented by the database.

[5] A lost update occurs when two unsynchronized transactions update the same value and both transactions have read the same original value. Hence, the update carried out by one of the transactions will effectively be lost.

[6] Atomicity, Consistency, Isolation, Durability.

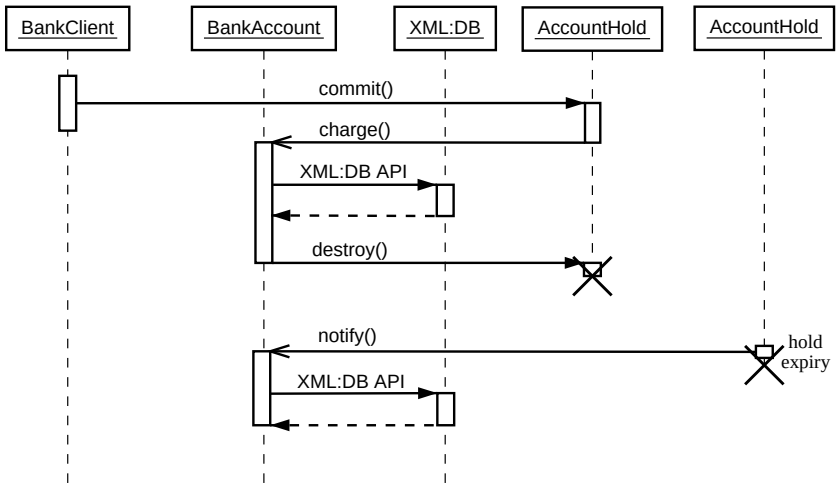

Figure 10: The use of callbacks and database calls.

6.2.3 Service Loaders

If a GT3 service container hosting a set of transient Grid services crashes, e.g., due to a bug in the service container code or a power failure, none of the transient services will be available after container restart. To resolve this problem, GT3 introduces the concept of a *service loader*. A service loader is responsible for recovering a service instance from secondary storage (or by any other means), dynamically deploying and activating the service in the service container. A service loader is called whenever an invocation is performed on a service which is not found in the container. It is then up to the service loader to find the invoked service, if it exists.

Both `BankAccount` and `AccountHold` service instances are (re)loaded from the database by service loaders. The service loaders make use of XML handlers to parse and recover the checkpointed service state.

Service loaders are only called on the first invocation of a service instance. Until that occurs, a restarted service container will have no knowledge of a previously stored service instance nor its state and lifetime. In the case of an `AccountHold` instance this is not acceptable, since the amount reserved by the hold must be reclaimed by the parent account on lifetime expiry. But if no client makes a call on the `AccountHold` instance, it will never be loaded and hence the reserved amount is locked away forever! Note that this is *not* an unlikely scenario since clients rely on the "soft-state semantics" of `AccountHold` instances. This problem can be resolved by loading all `AccountHold` instances on service container startup. The memory footprint of all loaded `AccountHold` instances can be minimized by loading them in a deactivated state.

6.2.4 Security Extensions

The GT3 framework provides a declarative security specification through the use of *security deployment descriptors*. Authentication method as well as handling of

delegated credentials can be specified on a per-operation basis. Disappointingly, the available framework for authorization was found to be too coarse, only allowing authorization to be specified on a per-service basis. We would like to be able to specify service authorization on an operation-level granularity. Specifically, different `BankAccount` users should have different access privileges. Regular users should only have access to the requestHold and findServiceData operations, whereas critical operations such as addAllocation and destroy should only be accessible by administrators. Since the current all-or-nothing access to a service is not enough for our purposes, we provide a more fine-grained authorization framework.

GT3 uses different handlers, which are server-side objects intercepting a SOAP request as it travels towards the service implementation. The AuthorizationHandler is responsible for finding an object implementing the ServiceAuthorization interface. That object is then delegated the responsibility for making the authorization decision. The AuthorizationHandler is a key component in our approach to fine-grained authorization, which we achieve through three components:

- Our *ServiceAuthorizationManagement* portType, allowing an ACL, holding a set of service users together with their individual allowed operations, to be associated with a service. `BankAccount` extends this portType.
- A customized *AuthorizationHandler* implementation. Our custom AuthorizationHandler dynamically loads a class implementing the ServiceAuthorization interface, if such a class is specified in the service configuration. Specifically, we can specify our own ServciceAuthorization implementation in the `BankAccount` configuration.
- *ServiceAuthorizationImpl*, a new class implementing the ServiceAuthorization interface. It bases the authorization decision on the requester credentials in the SOAP request, and only grants the requester access if he/she is a service user and the invoked operation is included in the allowed operations of that service user. This access control is performed using the serviceUsers SDE of the ServiceAuthorizationManagement portType.

Figure 11 illustrates the main interactions in this approach.

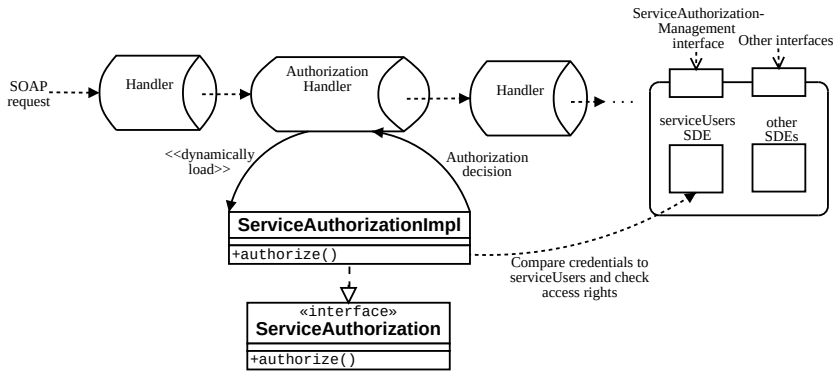

Figure 11: Achieving operation-level authorization.

The solution is flexible in that it, except allowing authorization to be specified

on a per-operation basis, allows different implementations of the ServiceAuthorization and ServiceAuthorizationManagement interfaces to be used, e.g., with improved performance or extended capabilities.

6.2.5 Threading Behavior and Testing

The GT3 service container is multi-threaded and as such a Grid service can potentially receive several concurrent client invocations at any time. Thus, thread synchronization becomes an issue. Access to shared resources, such as internal service state and the database, must be controlled. At the same time, the bank needs to be able to handle a potentially large number of simultaneous clients. Hence, synchronization must not be too coarse.

The thread safety of Xindice is not explicitly stated in the provided documentation. Furthermore, mailing list discussions provide little guidance on the thread-safe use of Xindice. Hence, our only option was to test it ourselves. To this end, stress tests have been developed where lots of threads are issued to bombard the bank with requests in order to uncover threading issues.

During these tests several Xindice threading issues seem to have been revealed. These issues have been resolved, and currently the stress tests exhibit the expected behavior. However, future end-to-end testing might uncover further issues.

7 Related Work

The area of Grid accounting has also been investigated by others. Some of these have provided guidance in outlining the accounting system architecture as well as in the definition of service interfaces. Two efforts which have had more substantial influence on SGAS are presented in this section. However, for different reasons, some of which are covered below, no single solution has been adopted in its entirety.

7.1 GGF Accounting Efforts

Efforts within the Global Grid Forum (GGF) are underway developing accounting related standards. Defined within the OGSA framework, these standards specify OGSI-compliant service interfaces useful in accounting. At the time of writing, there are no implementations available for any of these services. When there are, those implementations will be evaluated and considered for use in SGAS.

A Usage Record (UR) [21] defines a common format for exchanging basic accounting and usage data over the Grid. URs contain usage information which could potentially be collected at Grid sites, such as CPU-time, memory and network usage. A UR is an XML document whose structure is specified in a schema defined in terms of the XML Schema Definition (XSD) [37] language. The UR format has been adopted, in its entirety, by SGAS.

URs can be stored in a Resource Usage Service (RUS) [25], which is a Grid service for publishing and retrieving information about resource usage. The RUS is a conceptually simple logging service, since it is a write once/read many type of service. However, the RUS interface was found to be more complicated than necessary. E.g., the RUS interface makes use of a set of operations to update and query the URs, whereas this can be achieved through the findServiceData and setServiceData operations defined in the common OGSI:GridService interface. Thus, the RUS interface was not adopted by SGAS.

The UR and RUS efforts are mainly defined to support the Grid Economic Services Architecture (GESA) [24], which outlines service interfaces for enabling charging of Grid services in OGSA. GESA introduces the Chargeable Grid Service (CGS) and Grid Banking Service (GBS) concepts. A CGS encapsulates a regular Grid service which can be purchased, while a GBS registers the financial transactions involved in such interactions. The SGAS Bank component, and the service interface operations it specifies, have been influenced by the interfaces defined in GESA. Specifically, the hold concept origins from the GBSHold service of GESA. Although providing guidance to interface design, GBS was not adopted in its entirety, since it was not considered to make sufficient use of the capabilities provided by OGSI.

7.2 DataGrid Accounting System (DGAS)

The DataGrid Project [5] has developed an accounting system which has actually been implemented – the DataGrid Accounting System (DGAS) [1, 20]. SGAS shows much resemblance with DGAS in terms of the general architecture. Specifically, the one-bank-per-VO approach was influenced by the Home Location Register (HLR) concept of DGAS[7], which acts as a local bank branch managing the fund status of a subset of Grid users and resources.

There are several reasons why the DGAS software was not adopted by SGAS. First of all, DataGrid provides a Grid environment with rather special characteristics, which to a large extent is different from the SweGrid environment. E.g., DataGrid employs a centralized resource broker intercepting all job submissions within the Grid. Such centralized solutions are not in agreement with the decentralized nature of the Grid. DGAS is based on the concept of a *computational economy* and strives to achieve a self-regulating Grid market where supply and demand of resources in concert with dynamic price settings will lead to load balance among the Grid resources. By offering services, resource providers earn GridCredits which can then be spent by using other resources. This model seems a bit too elaborate for the needs of SGAS, and also a bit restricting in the general case since all resource providers may not want to be compensated in terms of resource usage. Furthermore, DGAS is not OGSI-compliant, but based on a custom designed, XML-based protocol. Such ad hoc protocols are less likely to reach widespread deployment.

[7] Which, in turn, DGAS has borrowed from the GSM network.

8 Future work

The next steps in the SGAS development include deployment of the SGAS prototype on a set of SweGrid clusters in order to perform end-to-end testing. The development of the production version of SGAS will be initiated, starting from scratch with a new code base. To this end, the experience gained in developing the prototype system will provide useful input. An alpha version of SGAS is scheduled for April.

Depending on the success of the deployment in terms of observed performance and the expected system load, measures might need to be taken to achieve scalability and load balancing in the banking system. Scalability issues can be resolved by distributing the bank service across several *sub-banks* – each responsible for a disjoint set of accounts. Since accounts are independent of each other, such a service distribution should be rather straight-forward to accomplish. GT3 offers a *virtual hosting environment* framework [31] which could be used for purposes of achieving scalability by spreading Grid service instances created by a factory to remote service containers.

Other topics which might be subject to further investigation include:

- Improving scalability by providing batch operations, issuing several operations in a single invocation.
- The use of replication to achieve fault tolerance and high availability.
- Incorporating additional resource types.
- Performing automatic price negotiations between Grid entities.
- Investigating issues related to service agreements, contract monitoring, fail-over and compensation.
- Transition to the Web Services Resource Framework (WSRF) [27], which is basically a refactoring of the OGSI specification into a generic and open framework for modeling and accessing stateful resources using Web Services.

9 Conclusion

This thesis proposes an accounting system architecture, suitable for computational Grids. Based on the proposed architecture, a proof-of-concept accounting system prototype (SGAS), satisfying the accounting needs of SweGrid, has been designed and implemented. This prototype will form the basis for a future production version and the experiences gained in developing the prototype will provide useful guidance in implementing the final version.

The accounting system leverages state-of-the-art Grid and Web Services technologies and offers transparency to Grid users, while providing fine-grained end-to-end security. Furthermore, the accounting system is based on open Grid standardization efforts (such as OGSA and OGSI) and provides a single point

of integration with underlying Grid middleware, allowing non-intrusive deployment in different Grid environments.

SGAS has been implemented using the Globus Toolkit 3 middleware, which provides an open-source reference implementation of OGSI. The main components of SGAS are modeled as Grid services. Grid services, as specified by OGSI, forms a stateful and transient class of Web services. By building SGAS on OGSA concepts, we believe that we have improved the chances of achieving more widespread adoption of SGAS, especially as OGSA is starting to gain strong industry support, and is widely seen as a standardization cornerstone within the Grid community.

Acknowledgments

I would like to thank my supervisor Erik Elmroth, for supporting me and providing guidance throughout the entire project. I would also like to thank Thomas Sandholm for a rewarding collaboration and for patiently answering my questions and providing me with valuable information around OGSA, OGSI and the GT3 framework. Others, which have provided comments and support along the way are Åke Sandgren and Olle Mulmo.

References

[1] C. Anglano, S. Barale, L. Gaido, A. Guarise, S. Lusso, and A. Werbrouck. An Accounting System for the DataGrid Project: Prelimiary Proposal – v3.1. Internet, February 2003. http://www.to.infn.it/grid/accounting/Current_prop.pdf.

[2] Apache Xindice, January 2004. http://xml.apache.org/xindice/.

[3] N. Bieberstein, C. Gilzean, and J.Y. Girard et al. *Enabling Applications for Grid Computing with Globus*, chapter 1. IBM Corp., 2003.

[4] LHC Computing Grid Project (LCG), January 2004. http://lcg.web.cern.ch/LCG/.

[5] The DataGrid Project, March 2004. http://www.eu-datagrid.org/.

[6] E. Elmroth and P. Gardfjäll. An OGSA-based Bank Service for Grid Accounting Systems. In J. Wasniewski et.al. (eds.), *Applied Parallel Computing. State-of-the-art in Scientific Computing*. Springer-Verlag, Lecture Notes in Computer Science (To appear).

[7] E. Elmroth, P. Gardfjäll, L. Johnsson, O. Mulmo, and T. Sandholm. An OGSA-based Accounting System for Allocation Enforcement Across HPC Centers. Submitted, February 2004.

[8] E. Elmroth, P. Gardfjäll, O. Mulmo, Å. Sandgren, and T. Sandholm. A Coordinated Accounting Solution for SweGrid. Internet, October 2003. http://www.pdc.kth.se/grid/sgas/docs/SGAS-0.1.3.pdf.

[9] Extensible Markup Language (XML), February 2004. http://www.w3.org/XML/.

[10] I. Foster. What is the Grid? A Three Point Checklist. *GRIDToday*, 2002.

[11] I. Foster and C. Kesselman. *The Grid: Blueprint for a Future Computing Infrastructure*, chapter 2. Morgan Kaufmann, 1998.

[12] I. Foster, C. Kesselman, J.M. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *IEEE Computer*, 35(6), 2002.

[13] I. Foster, C. Kesselman, J.M. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Internet, 2002. http://www.globus.org/research/papers/ogsa.pdf.

[14] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *ACM Conference on Computers and Security*, pages 83–91. ACM Press, 1998.

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

[16] I. Foster, S. Tuecke, et al. Open Grid Services Infrastructure: Version 1.0. Internet, June 2002. http://www.ggf.org/ogsi-wg.

[17] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[18] P. Gardfjäll. Design Document: SweGrid Accounting System Bank. Internet, December 2003. http://www.pdc.kth.se/grid/sgas/docs/SGAS-BANK-DD-0.1.pdf.

[19] Global Grid Forum, March 2004. http://www.ggf.org/.

[20] A. Guarise, A. Werbrouck, and R. Piro. DataGrid Accounting System: Architecture – v1.0. Internet, February 2003. http://www.to.infn.it/grid/accounting/Current_prop.pdf.

[21] S. Jackson and R. Lepro. Usage Record – XML Format. Internet, August 2003. http://www.psc.edu/~lfm/Grid/UR-WG/URWG-Schema.11.pdf.

[22] Java Remote Method Invocation, April 2004. http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html.

[23] NEESgrid, January 2004. http://www.neesgrid.org/.

[24] S. Newhouse. Grid Economic Services. Internet, June 2003. http://www.doc.ic.ac.uk/~sjn5/GGF/draft-ggf-gesa-services-1.pdf.

[25] S. Newhouse. Resource Usage Service (RUS). Internet, June 2003. http://www.doc.ic.ac.uk/~sjn5/GGF/draft-ggf-rus-service-1.pdf.

[26] NorduGrid, April 2004. http://www.nordugrid.org/.

[27] OASIS Web Services Resource Framework TC, April 2004. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf.

[28] OMG's CORBA Website, April 2004. http://www.corba.org/.

[29] T. Sandholm. Design Document: SweGrid Job Account Reservation Manager (JARM). Internet, November 2003. http://www.pdc.kth.se/grid/sgas/docs/SGAS-JARM-DD-0.1.pdf.

[30] T. Sandholm. Design Document: SweGrid Logging and Usage Tracking Service (LUTS). Internet, November 2003. http://www.pdc.kth.se/grid/sgas/docs/SGAS-LUTS-DD-0.1.pdf.

[31] T. Sandholm and J. Gawor. Globus Toolkit 3: A Grid Service Container Framework. Internet, July 2003. http://www-unix.globus.org/toolkit/3.0/ogsa/docs/gt3_core.pdf.

[32] SETI@home, January 2004. http://setiathome.ssl.berkeley.edu/.

[33] SNAC - Swedish National Allocations Committee, January 2004. http://www.snac.vr.se/.

[34] SOAP Specifications, February 2004. http://www.w3.org/TR/soap/.

[35] Swegrid, January 2004. http://www.swegrid.se/.

[36] The Globus Alliance, May 2004. http://www.globus.org/.

[37] W3C XML Schema, April 2004. http://www.w3.org/XML/Schema.

[38] Web Services, February 2004. http://www.w3.org/2002/ws/.

[39] Web Services Description Language (WSDL), February 2004. http://www.w3.org/TR/wsdl.

[40] WebServices – AXIS, January 2004. http://ws.apache.org/axis/.

[41] R. Butler Von Welch, D. Engbert, I. Foster, S. Tuecke, J. Volmer, and C. Kesselman. A National-scale Authentication Infrastructure. *IEEE Computer Society Press*, 2000.

[42] XML Path Language (XPath) Version 1.0, January 2004. http://www.w3.org/TR/xpath.

[43] XML:DB Initiative, January 2004. http://www.xmldb.org/.