A Coordinated Accounting Solution for SweGrid

Version: Draft 0.1.3 **Date:** October 7, 2003
Authors: Erik Elmroth
Peter Gardfjäll
Olle Mulmo
Åke Sandgren
Thomas Sandholm

Contents

1 Introduction

The primary aim of this document is to highlight issues that need to be considered in order to implement a coordinated accounting solution for the clusters within the SweGrid Grid. We also review some related research, to support the definition of a long-term strategy, and to identify areas that may be worth further in-depth investigation. The final aim of this document is to propose a high-level architecture that will guide future work on low-level design and proof of concept implementations.

2 Problem Statement

The main problem considered is to determine how a single resource allocation issued by the Swedish National Allocation Committee (SNAC) for SweGrid usage can be consumed and accounted for at any of the SweGrid clusters in a coordinated way. SNAC allocates computer time (*node hours*) in SweGrid for various scientific projects. These node hour allocations specify the node hours per month that each project is entitled to. Allocations are issued for a six or twelve month period at a time. Each month one sixth (or one twelfth depending on the length of allocation periods) of each project's node hours are withdrawn, unless the allocation has been consumed. Accounting is defined here as the phase in which the system audits the usage of system resources. The whole allocation may be used up at one cluster or partially at any number of clusters. The Grid users should not have to be concerned about which cluster runs his/her jobs (although some job descriptions may exclude all clusters but one). Which cluster is to run the job is determined at the time of submission by a workload manager (NorduGrid), and it can be based on static as well as dynamic system and job specification properties. Further, the allocations are not made to individual users but to projects. All local sites will hence need to have some minimal user-to-project mapping support in order to charge the correct account.

3 HPC Site Inventory

We have made an inventory at the local HPC sites that will administer resources in SweGrid in order to find out how they solve the accounting problem today. An overall goal is to design an architecture that can cooperate smoothly with the current local accounting systems with minimal intrusion. Furthermore, many of the problems faced and solved by the local sites will also have to be solved in the coordinated accounting layer.

The centers that were subject to the inventory include:

- HPC2N, Umeå
- Lunarc, Lund
- NSC, Linköping
- PDC, Stockholm
- UNICC, Göteborg [no answers received yet]
- Uppmax, Uppsala

We present their answers in the following categories: Schedulers, Usage Tracking, Project-User Mapping, Management Interfaces, Node Hours Mapping, Storage Allocation, and Account Management.

3.1 Schedulers

HPC2N

The batch system is Open PBS 2.3.16 (including some patches). This will probably be changed to Scalable PBS (i.e. Open PBS 2.3.12 and **a lot** of patches). Maui 3.2.6 is used for scheduling.

Lunarc

PBS Pro 5.1.4 and Maui 3.2.5 are used in a cluster with 193 nodes. Open PBS and Maui 3.2.5 are used in a cluster with 32 nodes. NQE is used on an SGI Origin 2000 system, which will soon be retired. Scalable Open PBS is considered for use in the future.

NSC

The batch systems used include OpenPBS, PBSpro, ScalablePBS and LSF. SGE (Sun Grid Engine) and SGEEE (SGE Enterprise Edition) are being considered for future use. For scheduling, MAUI and the job scheduler built into LSF are used.

PDC

A re-written version of Argonne Easy scheduler is used. The smallest atomic resource is a complete node or Unix/Linux machine. No concurrent jobs are allowed on any of the machines, mainly due to the lack of control of other aspects than wall-clock time in a standard Unix environment.

Uppmax

SGEEE (Sun Grid Engine Enterprise Edition) is used as scheduler and batch system today.

3.2 Usage Tracking

HPC2N

Currently, the following information is gathered concerning a job: executable, input files, output files, wall-clock time requested and used, and CPU usage. Disk and memory usage could probably be acquired as well but this would require some changes to the system and more/better tools.

Lunarc

The PBS accounting log files are used to extract usage information when composing statistical reports. On the SGI machine, the NQE accounting system is used to capture e.g. memory, disc, and cpu time information. [Editors Note: there is an effort within SourceForge to convert PBS accounting files into XML formats like those specified by the GGF UR working group. (See: http://pbsaccounting.sourceforge.net/).]
Number of used CPUs and CPU time are tracked for accounting purposes.

NSC

The information acquired for each job includes host, user, project/VO, the job time interval and CPU hours (wall-clock time).

PDC

Number of nodes used, resource kind (CPU, memory etc), and wall-clock time consumed are all tracked for accounting purposes. Resource kinds are based on increasing performance. (A faster

machine may be picked instead of a slower one if idling).

Uppmax
Information about CPU time, wall-clock time, memory usage, and i/o can be collected.

3.3 Project-User Mapping

HPC2N
Users are not mapped to projects. All NorduGrid users are mapped to the same local user account (n:1 mapping). The grid environment is currently "free of charge".

Lunarc
No project mappings are maintained. Only per-user information is tracked. The user group is relatively small (<100), which is why this model still scales. User group meetings are held three times a year to keep track of users.

NSC
Currently each user account is mapped against a single project.

PDC
A user is a member of one or more CACs (Charge Allocation Categories). The user can decide what CAC to charge when submitting a job. The CAC membership is then checked and validated by the workload manager before a slot is reserved to run the job. All users may also charge the 'free' CAC. A reservation charged to the free CAC has lower priority than any other reservation.

Uppmax
The mapping is maintained using regular files.

3.4 Management Interfaces

HPC2N
Currently, users can't retrieve any usage information except for used wall-clock time. Work is underway to create a user portal where all usage information could be collected.

Lunarc
Only standard Unix quota commands are used. Remote web interfaces are being developed for end users, but not for admin or management tasks. The PBS accounting files are transformed into a more readable text format that is used when viewing usage statistics. Note, wall-clock time is used in the reports, but CPUs and CPU time are used when giving out quotas. This is acceptable since no CPU quota violations are checked.

NSC
Allocation information is stored in a MySQL database. A web interface is available to search, display and modify accounting information. The web GUI can be used to update allocations, which are periodically synchronized against machines and schedulers.

Measures have recently been taken to start incorporating accounting information into this framework.

No web portal is currently available for users. Users can view their resource consumption for the latest month through commands on the resource of interest.

PDC
The information is stored in a flat ascii format. There are command line interfaces available to digest the information, and display it in a number of ways. Typically 'spsummary' and 'spjobsummary' could generate summaries or individual job-info, showing the charging sorted with respect to a mix of yearly/monthly/CAC/user/job, but also what resources were requested, when the resource was available, when it was released etc.

There is a also a web-based interface which simply outputs all CACs, granted resources, and how much was actually used each month. It is intended mainly for PDC staff monitoring the system. PDC mostly stores data in /afs/, which means that almost all PDC data is globally accessible, once you have proven your identity (or logged on) to AFS.

Uppmax
All the information is stored in regular files, and then accessed using the tools that come with SGEEE.

3.5 Node Hours Mapping

HPC2N
Node hours are converted to a percentage of the system over time. Using too much time lowers the user's/account's priority and vice versa.

Lunarc
No SNAC allocations have been issued for Lunarc, a mapping has therefore not been considered.

NSC
Node hour mappings differ between machines. Node hours are equivalent to user CPU hours on the SGI system, whereas Monolith and other clusters use wall-clock time.

Three priority levels exist; Normal, Bonus and Disabled. Jobs belonging to users who have not consumed their allocations are given Normal priority.
Jobs belonging to users having consumed their allocation are given Bonus priority in case the job is run on behalf of a SNAC project, or Disabled priority if the job is part of a test project. Normal priority jobs are given a great priority offset in MAUI, effectively making all Normal jobs run prior to Bonus jobs. Disabled jobs are not scheduled at all. The same costs are used for all priority levels. MAUI priorities are periodically updated from the MySQL database and from local accounting information.

Fair-share scheduling is not used, since it is more dynamic and makes it harder for users to understand system priorities.

Cluster jobs are not preempted. SMP nodes cannot be shared by two jobs. In the day-time standing reservations enables small, short duration jobs to be run.

PDC
SNAC Node hours are interpreted as wall-clock time and then charged to each CAC (no priorities apart from free vs project CAC exist). There is no enforcement - in practice there are projects that don't use their time, and others that are more active one month but less active the next. The ratio of node x time granted and the node x time actually consumed is monitored, and if a noticeable deviation to what is granted is observed the group of users will receive an e-mail. It is possible to introduce enforcement. It has however been concluded that it would not cause any net gain in resource utilization because of users irregular usage patterns across the allocation periods (currently per moth). A more important objective is to keep a high throughput of jobs of all CACs.

Uppmax
No SNAC allocations have been issued for Uppmax, a mapping has therefore not been considered.

3.6 Storage Allocation

HPC2N
Users get the amount of storage they request. The amount is allocated as AFS Volume Quota and cannot be exceeded by the users.

Lunarc
Storage allocations are controlled with regular unix quota. A default quota is assigned to every user, both in the home directory and in the scratch file system. Users can then request increased quotas.

NSC
Three storage areas exist:
1. /home (backed up NFS)
2. /disk/global (large backed up NFS)
3. /disk/local (local storage on each node; not backed up and cleared prior to job execution)

Even though there have been some problems, home should be controlled through quota. The /local area is only used by one user at a time.

PDC
Four different kinds of storage systems are utilized: 1) the home directory (always in /afs/), 2) scratch file systems, 3) parallel file systems and 4) HSM.

1) AFS
AFS is a global file system, accessible on any individual node/machine within and outside PDC. It is backed up on a daily basis. Quotas are enforced. Users have anywhere from a few hundred MB to 10s of GB in their AFS home directory. It is not a very high-speed file system.

2) Scratch

On each node, there is a node-local (local disk) /scratch/ file system, typically sizes from 4-30 GB. The local /scratch/ is available only during the reservation, and it is cleared after the reservation has completed. Any data on /scratch/ has to be moved to another storage or it will be removed by the batch system.

3) Parallel File Systems
A parallel file system typically is accessible and identical on any node within a certain system (set of nodes.) There are directories in e.g. /pfs/scratch/ where the scratch parallel file system is intended for temporary use (intermediate results, checkpoint/restart data etc.) Data on the *scratch parallel file system* is not necessarily backed up. To avoid filling the scratch parallel file system, files unreferenced for a certain time are removed. Typically files referenced within the past 7-21 days are not removed.
The *projects parallel file system* in e.g. /pfs/project/ is intended for more long-term storage of data - i.e. initial conditions, final results, reference results et cetera. There are projects parallel filesystems with quota, and projects parallel file systems without quota. The current thinking is that quota should be enforced, both with respect to the size and the number of files in the parallel file system. It has been sufficient to keep an regular eye on the projects parallel file systems without quota, emailing users that have a larger portion of recently unreferenced data on it, asking them to move it to the HSM system.

4) HSM
Data put on the HSM system finally, and automatically, is moved to tape (and transparently back from tape again, when accessed.) This is the final store for data not needed within the nearest future. The default quota is 50GB/user, but could be raised upon request.

Uppmax
No general strategy for data storage exists at this point. The users' home directories are used for long-term storage, because there are only local users.

3.7 Account Management

HPC2N
Follow the instructions at www.hpc2n.umu.se/account; sign the form, have the PI (Principal Investigator) sign it too, send it to the mentioned address and wait for mail notifying the user that the account has been opened. The account-webpages create a file with all the information needed to create the account. When the signed form arrives the account is created.

Lunarc
Individual local user accounts are created and maintained, and it is all that is required in order to submit jobs in the cluster.

NSC
The user specifies when he/she wants the account opened in an account application, which must include a copy of the user's passport.
On the SGI, a SecureID card and an introductory package (including a manual) is distributed to new users.

The process of transferring user allocation information from the MySQL database to the machines is mostly manual, although scripts exist to simplify the process.

PDC
Individual local user accounts are created for all users. But users can only submit jobs in the free CAC until they become registered in the CAC database, where SNAC identities are also maintained.

Uppmax
Uppmax is not a SNAC resource today, so no general account management strategy is in place for this purpose.

4 Related Work

There exist a number of related research projects developing or specifying technology relevant to a large-scale, heterogeneous, and distributed accounting system. In the following, some of the more significant contributions are presented.

4.1 Global Grid Forum (GGF)

There are four research groups within GGF [1] currently working on accounting related standards. They are all developed within the framework of the Open Grid Services Architecture (OGSA)[2], and in compliance with the Open Grid Services Infrastructure (OGSI)[3]. OGSA, and OGSI are also defined in the GGF. Currently, no implementations are available, but making sure that our solution follows the same approach and reuses data formats, and protocols where appropriate facilitates two things: 1) when off-the-shelf products based on these standards become available it will be easier for us to switch to using them, 2) because most future accounting systems for the Grid are likely to be developed with these standards in mind, it is an interoperability guarantee to let our solution follow a similar path.

Grid Resource Allocation Agreement Protocol (GRAAP)
The GRAAP [4] working group is currently specifying a Service Level Agreement (SLA) based protocol called Agreement-based Service Management (WS-Agreement) [5]. OGSI-Agreement defines the interactions involved in negotiating and reserving a resource usage contract between *agreement providers* and *agreement initiators*. The contract in turn specifies the Quality of Service (QoS) of a *delivered service* that the *service consumer* can expect. Further, a contract state machine is defined as well as a term-language framework. Some standard terms are defined but most terms are expected to be standardized in a domain specific context, e.g. for a particular job description language. In fact, unifying the plethora of job description languages used in the Grid today was one of the motivating factors behind the OGSI-Agreement effort, and one of its first applications. SweGrid initially uses NorduGrid XRSL as job description language, which is an extension of a subset of Globus RSL. Globus RSL is in turn being redesigned to comply with the OGSI-Agreement specified term language as well as a standard Job Submission Description Language (JSDL) [6].

Resource Usage Service (RUS), and Usage Record (UR)
RUS [7], is a service that can be used to publish and query resource usage data. It relies heavily on the Usage Record format [8], which is a standard XML document composed of the various

usage properties like CPU-time, wall-clock time, and disk space, which Grid resources may record. RUS, is hence intended to be used both by resources and by various brokers and users interested in usage information, e.g., banking services, work load managers and resource funding agencies. RUS is a fairly trivial information publishing and retrieval service, which should be easy to incorporate into the SweGrid system provided that the usage data logs are based on the UR defined XML format.

Grid Economic Services Architecture (GESA)
GESA [9] essentially extends the OGSI Grid service model into an economic service model, where you can charge consumers for service usage. In order to achieve this GESA defines an architecture based on OGSI-Agreement contracts, and Resource Usage services. This combines all accounting related efforts within GGF into a common model. Two new components are also defined, a Grid banking service and a chargeable Grid service, the latter being a direct extension of the OGSI Grid service.

In summary, there are several efforts within the GGF relevant to our work. GESA may provide some architectural guidance to our solution, but initially the most important specification to follow is the Usage Record, since it defines a common data model among the various accounting components. When NorduGrid is upgraded to Globus Toolkit 3 the GRAAP work also becomes relevant.

4.2 European Grid Projects

There are a number of Grid Projects within the EU that are considering various accounting systems to fit their needs. An inventory recently made by the GridStart projects in the EU [10], however, showed that no complete solutions are yet in production use. The most promising work on a more generic Grid resource accounting model has been developed by the DataGrid project, and is called the Data Grid Accounting System (DGAS).

GridStart
AVO [11], EGSO [12], and FlowGrid [13] all have similar accounting needs to SweGrid's (based on scientific funding, and usage tracking focused), whereas EuroGrid [14], GRIA [15], GEMSS [16], and GRASP [17] take more of a business model approach similar to the GESA chargeable service model with an emphasis on the billing process [10]. However, since most of the accounting systems relevant to the SweGrid environment are likely to converge around the most elaborate system to date, DGAS, we have focused our investigation on that particular system.

DGAS
The DGAS [18] model envisions a whole new economic Grid market, where supply and demand of Grid resources work in unison to strive towards equilibrium where all resources are fully utilized to the lowest possible price. This equilibrium is achieved by a built-in, self-adapting feedback mechanism, where GridCredits can be earned by offering resources, and spent by using resources, and thus mimicking the monetary market. Although visionary and elaborate this model has some shortcomings, an obvious one being that some resource providers would want to trade their earned Grid credits against something else than resource usage. Even though the underlying economic model of DGAS might not be a perfect fit for SweGrid, it is a very good fit on an

architectural and technical level, as we shall see.

One of the nicest features of DGAS is that it is fully distributed and thus true to the decentralized cornerstone idea of the Grid. Each user has an account in a local bank called the Home Location Registry (HLR). When a job is submitted by the user, the resource broker receiving the request contacts a pricing authority at various resources and the local bank to check whether there are sufficient funds to run the job. If the bank grants the transaction the request is passed to the job controller which sends it to an available resource that matched the user requirements. A resource monitoring service then tracks the job status and the resource usage and sends periodic reports to the HLR. When the job completes the total cost is calculated and possible holds on amounts in the HLR are unlocked and the credits spent are withdrawn from the user HLR and deposited into an account in the resource HLR. The only part of this interaction that may not have any equivalent action in a SweGrid scenario is the Resource HLR transaction. The rest of the interaction seems relevant, and there is already an initial implementation available of DGAS based on Globus, which should by evaluated further. We have two concerns with DGAS, though, 1) it may be a bit too elaborate and complex for a simple SweGrid job submission, 2) it is not based on OGSA technologies. However, an evaluation of the implementation might still be valuable to gain understanding of the problem domain.

4.3 Other Grid Projects

Nimrod/G [19] implements a novice cost/weight model where price matrices of resource costs for particular users are defined and used by workload managers to for instance minimize the cost for a particular user that submits the request. The heuristic model allows resources to balance the load among them by changing the weights of the costs for particular users depending on the current system usage. Further, resource monitors update performance rates and publish the information to workload managers to allow soft performance guarantees based on a the dynamic pricing model.

SNUPI [20] is a performance and usage monitoring toolkit for Linux clusters, which was developed to easily obtain, store and query resource usage information in a Grid. Usage data is stored in relational databases and can then be queried via web Portals. SNUPI is interesting in that it shows how scheduling agnostic resource usage tracking and information publishing can be done with simple extensions to the Unix monitoring system built into Linux.

5 Summary of Issues

[21] outlines some issues that need to be solved by distributed accounting systems on the Grid. Here we discuss some of the issues most relevant to our SweGrid environment.

5.1 Dynamic Account Management

When a job is submitted to the SweGrid it could potentially be executed on any of the participating cluster systems that meet the requirements of the job description. This means that the user would need to have a local account at all of the possible clusters. As the number of users and clusters grow, manual creation of these accounts becomes a serious scalability as well as resource consumption issue. The current version of NorduGrid only executes jobs at sites where a local account exists; this could however invalidate the entire workload distribution, which is

central to the Grid architecture. There are two possible strategies when it comes to dynamic account management 1) a pool of reusable accounts with expiration timestamps could be set up a priory by a system admin, 2) accounts could be created dynamically when resources become scarce by a super user broker. Approach 1) has the disadvantage that it still requires authorization rights to be set up for each new user at runtime. Approach 2) is more flexible but requires privileged user rights of the broker component that sets up the account and is thus more vulnerable for attacks. Some combination of 1) and 2) is also a viable option. Strategy 1 has been implemented for Globus Toolkit 2 in [22], whereas strategy 2 has been prototyped with Globus Toolkit 3 in [23]. Strategy 2 seems to be the best fit for SweGrid although it is a bit more elaborate to deploy.

5.2 Allocation Violation Enforcement

Excess of the wall-clock time requirement of a job is the easiest contract violation to monitor, and react to. With a virtual machine approach as in [23] more elaborate contracts could be set up and monitored. Initially detection of exceeded wall-clock time or SNAC node hours seem to be enough for SweGrid. It must be clearly communicated to (or alternatively negotiated with) the users through published policies how various resources react to these violations. This is a wide field of ongoing research and SweGrid could provide an interesting testbed for these SLA negotiations in the future.

5.3 Project-User Mapping

Allocations are made to projects not individual users. When a user submits a job it must therefore always specify what project to charge the submission to. In a generic broker scenario this is best communicated through the job description (XRSL in NorduGrid). The project to user mapping could be dynamic in which case a VO Management service may be a good approach. VOMS [24], and CAS[25] are two possible solutions when it comes to setting up and granting projects and project members various rights dynamically. Initially in Swegird an elaborate VO Management solution may be overkill, but there must be a consistent way for all the resources in SweGrid to check user-project mappings dynamically (compare with the dynamic account issue), through some (semi) on-line service. Currently all user-project mappings are local to a cluster site, which is why this problem did not surface before the SweGrid was set up.

5.4 Consistent Usage Record

Usage records must be obtained at each local resource site, and presented in a consistent way to the accounting system. The GGF UR work seems to be the most appropriate standard record format, which could be leveraged in SweGrid.

5.5 On Line/Off Line Centralized/Decentralized “Banking” Service

When implementing a distributed accounting solution consistency becomes a non-trivial issue. Large-scale deployments have shown that strict transactional consistency may not scale very well in a Grid [add Globus MDS 1, MDS 2 ref here]. A weaker consistency strategy may hence be more appropriate. This could for example be done with off-line periodic synchronization, or on-line caching based polling or pushing of information. The important thing is to make the system appear as if being strictly consistent to the end users (this is very important in our bank scenario when funds are being exchanged). There is also a risk of ending up with a centralized bottleneck when consistency is maintained to strictly. Distribution of allocation information and comparison

with local usage information (note could be both polling and pushing) is a good example of a case where weak consistency might be implemented. Distributed banking services might synchronize with more central banking services periodically. Note this could be done in a hierarchical way similar to how Globus MDS is designed. In summary both scalability and availability can be improved by not using a consistency model that guarantees the ACID properties at any point in time. More elaborate transactional models may however be investigated, as the WS-Transaction work [WS-Transaction ref here] matures.

5.6 SNAC Allocation Format

SNAC allocations are currently modelled after single site allocations. For SweGrid this has some apparent drawbacks, e.g. when it comes to mapping local user accounts to allocations. The SNAC format might also benefit from following a more standard SLA approach like OGSI-Agreement or WS-Policy, so that it can be fed straight into QoS aware brokers in the future.

6 SweGrid Accounting System Architecture Proposal

This section covers the proposed architecture of the SweGrid Accounting System (SGAS). Before presenting the architecture we list some of our requirements on the system and cover some of the design issues together with the choices made as well as a discussion of alternatives and specific problems that need to be addressed. We also outline areas requiring further investigation, which might be interesting for future versions. Besides presenting an overview of the proposed architecture, this section also motivates some of the design choices.

6.1 Functional Requirements

In any Grid environment, resource providers (i.e. *resources*) and resource consumers (i.e. *users*) are the main entities interacting with each other. Note that in the SweGrid context, a resource is equivalent to a cluster. Users request the services provided by resources. This makes users and resources a good starting point for making a list of the minimal functionality required from the accounting system.

From a resource's perspective, the accounting system needs some means to:

- Provide cost information to users
- Grant/deny users access based on account balance (i.e. authorization is based on ability to pay)
- Get a guarantee that user funds will be available to pay for resource usage
- Track resource usage
- Charge the user for the resource usage
- Get resource usage information for each submitted job (e.g. for evaluation purposes)

From a user's perspective (or some entity working on behalf of the user), the accounting system needs some means to:

- Specify which VO/project should be charged for resource usage
- Get the cost associated with using a resource
- Get account balance

- Get information on completed transactions
- Get usage information on completed jobs

6.2 System Properties

The accounting system should possess several important properties. Some of the most critical properties and their implications on the architecture are listed in this section. A more detailed description of the architecture is presented in section 6.4.

6.2.1 System Scalability

The accounting system needs to scale with an increasing number of users and resources. This outrules any centralized solution where all Grid accounts are handled by the same entity, as such an architecture cannot be made scalable.

Our architecture does not assume any centralized account handling entity. (We refer to an account handling entity as a *bank.*) In our architecture each VO has an associated bank which handles the accounts of the VO users. In case a VO grows too large (and scalability is becoming an issue) the VO could either be split into two VOs with two separate bank services, or the VO could deploy a second bank.

Another scalability issue is that of user accounts, which has also been covered in a previous section (5.1). The current practice is that in order for a grid user to execute jobs on a resource, the user needs a local account on that resource. Grid users are mapped to local accounts either using a 1:1 or an n:1 mapping. The 1:1 mapping does not scale with many users and the n:1 approach might make it hard to track individual user's resource usage and users might interfere with each other. Also, with n:1 mapping all users have the same access rights, hence violating basic privacy requirements. In a general accounting system, a more flexible approach for user mapping is needed. An interesting alternative is based on dynamic creation of user accounts (so called *template accounts*) at job submission time (see section 5.1) but an initial SweGrid implementation might not incorporate such features. However, their potential and possible future use will be investigated.

6.2.2 User Transparency

In an ideal accounting system, the end user would not notice any difference when the accounting system is deployed. This is probably too optimistic in many cases; e.g., a user participating in multiple projects will at least need to specify what project should be charged for the resource usage. We would also like to provide client-side programmer transparency. I.e., we would like to keep the client-side as "clean" (unmodified) as possible. A program (e.g. a globus program) 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. This will be apparent from the architecture overview (section 6.4).

6.2.3 Consistency

There must be a consistent view of account(ing) information throughout the entire system. E.g., a user with 1000 "credits" left on her account should not be able to submit six 1000-credit jobs concurrently to different resources and have them all run.

The proposed architecture provides consistency during normal circumstances. In case of failure of some critical entity (such as a bank) less strict levels of consistency can (optionally) be accepted. Loose consistency can be governed by various caching strategies, where for e.g. cached information can be read but not used for updates (optimistic locking).

6.2.4 Fault Tolerance

The accounting system must not have any single-point-of-failure. The system must run smoothly even in the event of component failures. It is critical that the accounting system exhibits a high level of availability, otherwise consistency cannot be guaranteed. Consequently, some of the key accounting system entities probably need to be replicated in order to provide a highly available system.

Despite the use of replication failures still occur. In case of an unreachable bank some soft failure handling is needed. It would not be acceptable to deny all job submissions for a VO's users when the bank is unreachable. In such cases users should still be granted access to resources but with less strict consistency guarantees.

A first prototype of the SweGrid accounting system will probably not include replication, although it could be introduced later. The system will try to mask failures by proceeding as if nothing had happened, resolving inconsistencies at a later time.

6.2.5 Trust and Security

Accounting information should only be exchanged between trusted entities. This implies heavy use of authentication and authorization techniques. Access to accounting information should be limited to trusted entities (such as trusted users, their VOs, trusted resource providers, etc).

An initial accounting system for SweGrid will assume pre-existing trust relationships among the entities involved in a service exchange. However, we also investigate a more distrustful scenario and discuss a potential solution.

6.2.6 Other Requirements

Some other important requirements on the accounting system include handling of concurrent requests (while still maintaining consistency) and support for multi-site jobs.

6.3 Design Issues

There are several ways of realizing an accounting system. The proposed design can be referred to as an *online bank* approach. In this approach, a user is granted/denied access to resources based on the amount of available funds on his/her account. If a user submits a job without having sufficient funds available, the accounting system may not allow the job to run[1].

Other solutions, with less restrictive consistency demands, are also conceivable. An *offline bank* (where accounting information is cached locally at resources and then periodically synchronized with a bank) and a *gather/scatter* approach (where allocations are spread across resources and resource consumption information is gathered from all resources when needed) would offer alternative solutions.

[1] The system will probably allow for some temporary exceeding of funds, at least in an initial deployment phase.

Even though the overall idea for the accounting system has been decided, there are lots of open design issues, which need to be addressed. Before presenting the architecture overview, some of these design issues are covered, as well as the (preliminary) solutions chosen for SweGrid together with a discussion of possible future extensions.

6.3.1 Types of Resource Usage

What types of resource usage should the user be charged for? This, of course, depends on what resource usage information can be acquired, as well as the needs of the resource provider. All resource providers will not charge their users for all resource types. Here is a list of resource types that might be of interest:

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

In SweGrid, only wall-clock time (so called *node-hours*) is considered, but as the Grid evolves other resource usage types could be of interest as well.

6.3.2 The SweGrid Currency

An interesting issue is what "currency" should be used to charge a user. At least two alternatives exist:

- Charge for usage of each different resource
- Convert different types of resource usage into a single currency unit (such as *grid credits*). This approach generally requires some function to map a set of resource usage records into a single currency.

In SweGrid, the latter alternative is preferable. Since users are only charged for node-hours, no conversion function is needed. In the future, it may be of interest to use a combination of the two if other than SweGrid-specific resources are brought into SweGrid, i.e., resources for which the SNAC-allocations are dedicated.

6.3.3 Resource Valuation

Should resources somehow be valued (e.g. by performance)? One could argue that it should be less expensive to run a 2 hour job on a Pentium III than on a 486. Furthermore, in the case of all resource usage being converted into a single currency, the different resources need to be given appropriate weights.

In an initial accounting system for SweGrid, the resources (i.e. clusters) will probably be

uniformly valued. A more mature accounting system would allow resource providers to set cost weights on their resources and make those prices available to users, creating a kind of "grid market place".

6.3.4 Job-cost Information

If different resources are valued differently, a user must be able to obtain correct cost information in order to make an informed decision on the choice of resources. Some potential solutions are:

- Use static prices throughout the system.
- Query the resource.
- Make cost information available through some information system (such as MDS).

The first alternative is the most likely in an early SweGrid solution. A long-term solution should incorporate (at least) one of the other alternatives.

6.3.5 When to Charge the User

When should the user be charged for the resource usage? At least three alternatives exist:

- *pre-payment*: pay before usage
- *continuous-payment*: pay for used resources as job executes
- *post-payment*: pay after usage ("credit card model")

The pre-payment approach has the apparent problem of estimating an exact run time prior to job submission. The continuous-payment approach will probably place a heavy load on the accounting system, making it unscalable.

For SweGrid, we think that the last alternative is the only viable option. This approach relies on some means of reserving funds from a user's account. The resource must be guaranteed that sufficient funds will be available after the job has finished. Initially, the accounting system will probably have a somewhat less strict enforcement of the sufficient-funds-criteria, allowing a slight temporary exceeding of funds.

6.3.6 Failures

What happens in case of failures? E.g., if a resource provider crashes, the network fails, a user cancels a (queued or executing) job, etc. In such cases policies are needed to govern what actions need to be taken.

6.3.7 Logging and Resource Usage Tracking

It is likely that resource providers, VOs, allocations committees and users are interested in detailed information regarding job execution and resource usage. This information could, e.g., be used for evaluating resource usage or presenting information to the allocations committee, funding agencies, etc. Hence, some sort of logging service, capable of holding detailed information on resource usage might be needed. Moreover, from a user perspective, 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 the jobs (whether the job finished, failed, cancelled, etc), job input data and job output

data.

6.4 Architecture Overview

This section will present an overview of the architecture. The entities involved in the system and their relationships, together with an illustration of the accounting system interactions are presented.

6.4.1 Entities

The accounting system contains the following main entities:

- Users
- Virtual Organizations (VOs)
- Projects
- Resources
- Bank services
- Log services

In SweGrid the bank accounts will be assigned to projects and not to the individual project members, even though the members have individual user accounts on the Grid. Within each VO, a user might participate in multiple projects (with different bank accounts). Furthermore, there is nothing stopping a user from being a member of projects in several VOs.

Each VO has one associated bank service to manage the accounts of the VO projects. However, one bank might service several different VOs. I.e., there is a one-to-many mapping from a VO to its associated bank.

This one-bank-per-VO limit might be too restrictive in a general grid (accounting) environment. However, to date, we have not come up with a scenario where a VO would actually need the services of more than one bank. One such scenario might be if a VO grows too large. In that case a second bank could be introduced to offer some degree of load-balancing. Another such scenario is that of an allocation provider, such as SNAC, which does not trust the VO bank and would like to deploy its own bank. In such a case, it should probably set up its own VO rather than deploying a second bank for the existing VO. The use of more than one bank for a VO needs to be investigated further.

The log service holds detailed information about the resources consumed (Usage Records - URs), job status, input data, output data, etc. about jobs submitted by the VO users. The log is a conceptually simple service as it is a write-once/read-many kind of service.

6.4.2 Entity Interactions

The following illustration (see figure 1) outlines the main entities involved in the accounting system as well as their interactions during a typical job submission:

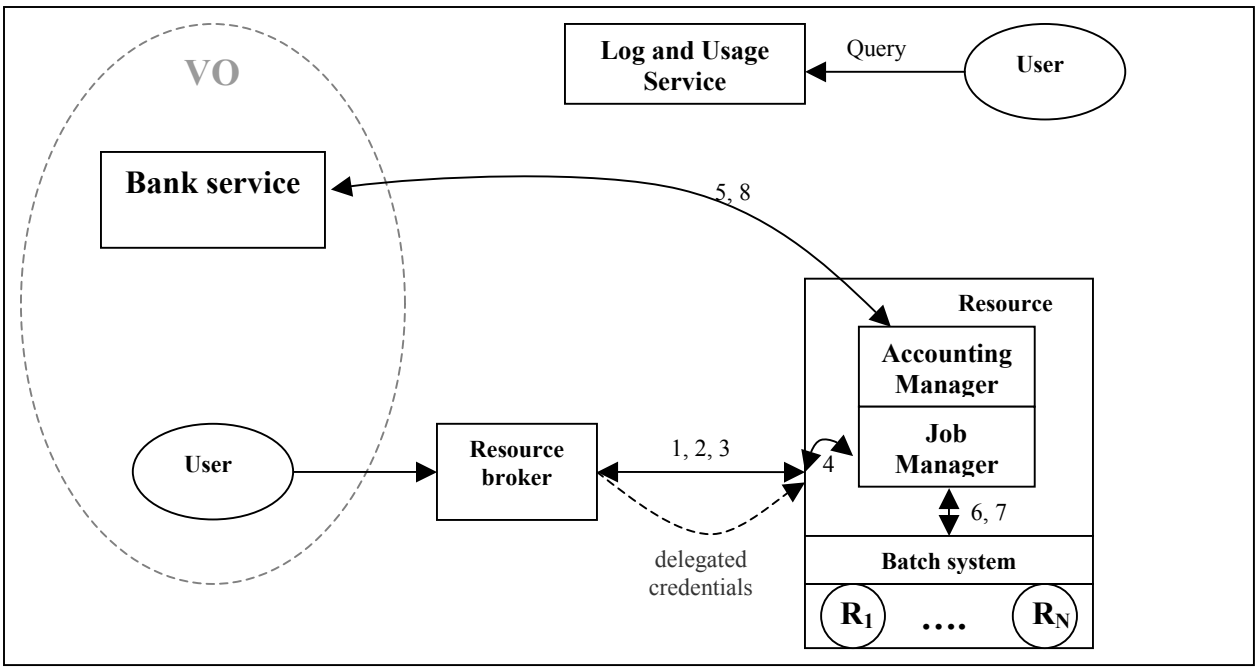

Figure 1. *Interactions among accounting system entities.*

1. The user (or an entity acting on behalf of user, such as a broker) contacts a resource that has been selected to execute the job.
2. The user and the resource mutually authenticate and the resource then authorizes user. The user's credentials are delegated to the resource, making the resource capable of authenticating on behalf of the user.
3. User submits a job request to the resource. The job request includes information about the VO (and its bank service) and project to be charged.
4. An accounting manager (AM) controlling a job manager (JM) is started to manage the job. The user's delegated credentials as well as the job request are forwarded to the AM. The AM ensures that the specified VO bank is trusted before proceeding, and forwarding the request to the JM.
5. The AM acquires a time-limited (soft-state) hold on an amount of the project's funds. The hold acts as a reservation on the project funds, guaranteeing the AM that sufficient funds will be available when the job finishes. The amount reserved is based on the run-time approximation in the user's job request. The AM mutually authenticates to the bank on the user's behalf using the delegated credentials. The bank only grants the hold if the user is recognized as a trusted user and sufficient funds are available in the project account. If the hold could be acquired a *hold-id* is returned to the AM. The hold-id is used later by the AM to refer to the specific hold.
6. If the hold is granted the JM 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 life-time of the hold.
7. After the job has completed (or cancelled) the AM gathers information about the resources consumed by the user's job and records this in a Usage Record (UR).
8. The AM charges the project account for the resource usage. The AM sends the hold-id together with the UR to the bank service. The bank charges the project account and the

hold on the project's funds is released. Any residual amount of the hold is simply returned to the account.

The bank service as well as the log service can be queried by users. E.g., a user might query the bank to get a list of its latest transactions. For more complete resource usage information about particular jobs a query can be posed to the log service. A bank transaction entry and a log entry could be associated by using the same system-wide unique job id.

6.4.3 Architecture Issues

This architecture has some issues which need to be solved, or at least taken into consideration.

6.4.3.1 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 (time-out) approach. With hard-state holds, where explicit "release hold" operations are needed, a resource crash after acquiring a hold could have the devastating effect of locking up an amount of a project's funds for good. A problem of using time-limited holds is that estimating the job runtime (including queue time) might be hard, before the job has been submitted to the batch system. Hence, some means must exist to extend the life-time of a hold after its acquiring.

6.4.3.2 Trust and Service Agreements

The architecture proposed here assumes pre-existing trust relationships between the involved entities. The resource only authorizes users of trusted VOs, and never makes deals with untrusted banks. The bank does not need to authorize the resource when making transactions, since the resource is acting on the user's behalf (using user's delegated credentials). This means that the user must be careful not to submit jobs (and hence credentials) to non-trusted resources. Having done that, nothing protects the user's account from a fraudulent resource. A similar approach which does not rely on the use of delegated credentials would be to have the bank keep a list of trusted resources, and giving these resources full rights to update the bank accounts. We refer to these approaches to service agreements, where a user blindly trusts the resource to act on its behalf, as *in-blanco signing*[2] approaches. The main motivation behind the in-blanco signing approach is the transparency it offers to the users. The presented architecture in figure 1 makes use of the in-blanco signing approach.

The in-blanco model is probably adequate for use in an initial implementation of the SweGrid accounting system. In a more general accounting system, both the user and the resource must come to a mutual agreement about the terms of the service exchange. This implies having both parties sign some form of *contract*, and the bank only granting the resource a hold if a valid contract can be shown. The contract would at least specify the resource prices to be used, as well as the approximated run time (effectively setting an upper bound on the total job cost). We refer to such an approach to service agreement, 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 (which is the default behavior).

[2] The user implicitly "signs" a contract giving the resource full access to the user's account.

The interactions in the contract signing approach could be as illustrated in figure 2. Note that some interactions have been left out in the illustration, which focuses on the use of contracts:

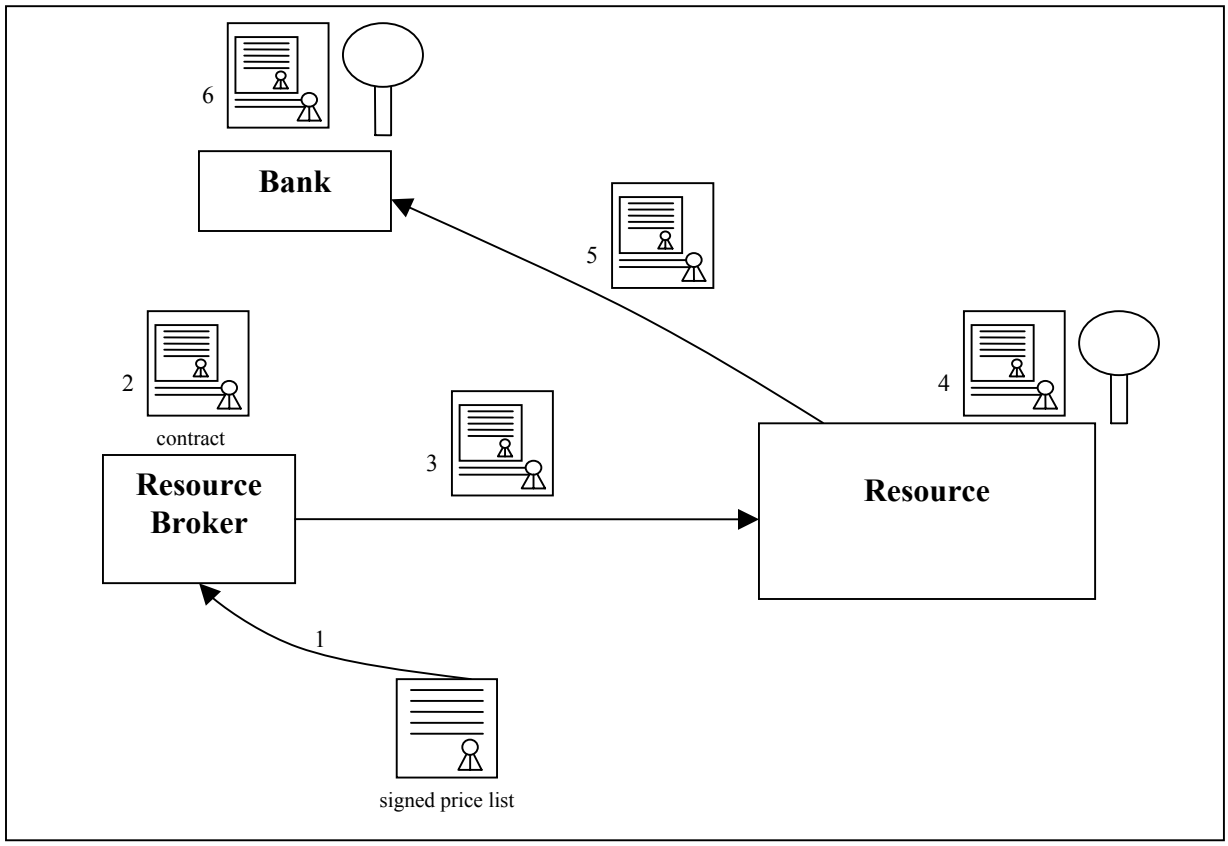

Figure 2. *Interactions in a contract signing service agreement.*

1. The user gets a *signed* price list from the resource. The price list could, e.g., be requested directly from the resource or be published in some information system, such as MDS.
2. If the prices are acceptable, the user appends the approximated run time (or entire RSL) to the price list and signs the entire document. The document is now a valid service agreement *contract*.
3. The contract is sent to the resource together with the job request (if it is not included in the contract).
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's 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 the 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.

6.4.3.3 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. The job-submission should be atomic in the sense that either all jobs are successfully submitted or none is. In co-allocation, the following scenario is not acceptable. A grid user submits ten (co-allocated) jobs to different resources. These jobs will run together as a unit, periodically exchanging data. The first nine jobs are successfully submitted but the tenth job cannot be started, due to a lack of user funds. Meanwhile, the nine jobs already started are consuming resources, without getting any useful work done, while waiting for the tenth job to join.

In such a situation we prefer not to put all responsibility on the user for keeping track of the funds available before submitting the job. Even if the user makes sure necessary funds are available for all jobs, some of these might not be available at run time. Another user of the project might submit a job meanwhile. From the user's perspective, there should be no difference in submitting ten jobs to be run at one site or have them all run on distinct resources. The handling of co-allocation should be transparent to the user.

Of course, the accounting system should not solve the problem of co-allocations (after all, this is a problem for the co-allocator), however we must make sure that the accounting system provides sufficient support to enable co-allocation.

To enable support for co-allocation, at least two approaches are conceivable. The first approach is based on extending/modifying the job submission protocol to perform some kind of *two-phase commit* submission (i.e. submitted jobs acquire holds, report back to the co-alloctor and then wait for a "commence" signal before they are submitted to the batch system). The Globus Toolkit (version 2 and 3) include a `two_phase` RSL attribute which might be used for such purposes.

Not all Grid middleware use a "GRAM-like" job submission protocol (e.g. NorduGrid). In such cases, an alternative solution could be to have the co-allocator acquire holds for all jobs it is about to submit. These pre-acquired holds could then be handed over to the resources responsible for executing the jobs, as part of the job submission (e.g. a hold-id could be passed in an RSL attribute).

For an initial version of SGAS, if we assume a less strict enforcement of the sufficient-funds-requirement (i.e., we can allow a user to temporarily exceed their available funds) and if we assume that the co-allocation feature will not be used too extensively, we can probably ignore this problem.

6.4.3.4 Fault-tolerance

Maintaining strict accounting consistency throughout the system requires the bank and log services to be reachable at all times. If the bank is not available, the resource cannot make sure that sufficient funds are available in the user's account. Hence, high availability of key entities is critical to ensuring consistency. The implementer has great freedom of choice in implementing these services (as long as they implement the minimum service interfaces), but since high availability is such a major concern measures for achieving some degree of fault-tolerance probably need to be taken. One potential solution is the use of replicated services. As already

mentioned, such a feature will probably not be a part of a first version of the accounting system. There are strict consistency demands on the bank. If the bank service is replicated great care needs to be taken to guarantee consistency between the replicas. The performance as well as the feasibility of such an approach would need to be investigated further.

Failures occur even in replicated systems. Despite a highly available system, some soft failure handling procedures are needed in case disaster strikes. The solution in our architecture is to mask system failures by accepting job submission even though bank contact cannot be established. The accounting information gathered from the job will be cached locally until a connection to the bank can be established again. When the bank service becomes available, the locally cached information is synchronized with the bank. While the bank is unavailable, locally cached information could also be used to discover exceeding of funds. This, however, requires the cached information to be sufficiently up to date. Of course, strict consistency is compromised during the failure handling. A user, being afraid of exceeding the account, should optionally be able to specify that the resource should not run the job unless contact with the bank can be established.

6.4.3.5 Log Service Placement

The observant reader has probably discovered that the log and usage service seems to be "disconnected" from the rest of the architecture (see figure 1). Currently, the architecture does not specify the placement of the log (i.e., where the actual logging is performed). The log and usage service is currently specified as an interface to the log, independent of the location of the log. This means that the architecture currently leaves the placement of the log to the accounting system implementer. For the log placement several alternatives (or any combination of them) could be considered:

- Logging is performed at bank
- Logging is performed on (each) resource
- The log is a separate service

When logging is performed at the bank or on each resource, the log and usage service becomes a read-only service, acting as an interface to the physical log. In the case of logging being performed by a separate service, the log service interface further needs to specify update operations for adding log entries.

There are arguments for each of these locations for the log and usage service. The bank will get the UR anyway (as part of charging the user's project) so it might perform the logging as well. Furthermore, having all logs in the bank service will make all logs related to the VO accessible from one single location. Performing logging on each resource will ensure that the resource manager has all usage information about the jobs run on its resource. This information is probably sufficient from the resource manager's perspective, who is probably only interested in information regarding the resource. On the other hand, the log service would need to gather information from several different log locations that might cause both poor performance and consistency problems (in case some resource log is unavailable). Performing logging in a separate service gives a modular solution that also works fine in the case where a bank is not needed (an accounting system that only *logs* resource usage). To let the log be a service of its

own raises some questions. Should there be several log services (e.g. one per VO) and who should update the log service (bank? resource? user?)?

The log and usage service collects both resource usage (e.g. like RUS), and job tracking (holding detailed information about completed jobs). Whether this means that two logs are maintained or that simply two interfaces to the same log are provided is an implementation detail not considered here. The resource usage interface should comply with the GGF RUS specification where appropriate. The logging interface is on the other hand specific to the SweGrid environment and it should be designed to complement with features not covered by RUS.

6.4.3.6 Security

The bank and log services needs to maintain some notion of trusted users (e.g. users having an account in bank) and privileged users (e.g. system administrators and trusted resources) and their access rights. The services will only allow access to entities who are capable of authenticating themselves as trusted or privileged users.

6.5 Service Interfaces

To make the accounting system interoperable and to allow some freedom of implementation to the developer, protocols are needed to interface with the bank and logging services. We intend to base SGAS on OGSA and all interfaces should be designed using OGSI compatible WSDL. Appendix A has a first proposal of a Bank service interface. [TODO: Move this from appendix to design document with fully motivated operations and XML/WSDL syntax]

6.6 Interoperability

We will investigate the possibility for interoperating with existing accounting system solutions, such as the EU DataGrid's DGAS. Would it be possible for our systems to co-operate, and in that case what is needed in order to achieve compatibility between the systems? Furthermore, we will keep an eye on the standardization activities within this area, e.g. the GGF-GESA working group.

7 Conclusion

We believe that building on the design of DGAS, and GESA with decentralized banking, and logging and usage tracking services responsible for collecting and debiting for resource usage is the best approach forward for a coordinated accounting solution in SweGrid. We further think that OGSA and OGSI are well suited to be used as a protocol framework for the envisioned services. Minimal impact is enforced on the local sites, by only requiring a small subset of the GGF Usage Record to be collected (initially only wall-clock time, because it maps best to SNAC node hours)

8 Future Work

TODO:
Drop or collect info from remaining HPC sites
Flesh out conclusion
Consolidate Issues Lists in this document
Compile UR subset that all centers must collect
Revisit SNAC allocation XML
Write low-level design document outlining phased implementation strategies

Specify all Service interfaces in WSDL
Evaluate DGAS implementation more thoroughly
Start Prototyping with GT3/NorduGrid

9 Acknowledgements

We would like to thank Lars Malinowsky, PDC, Stockholm; Per-Anders Wernberg, Lunarc, Lund; Leif Nixon, and Niclas Andersson, NSC, Linköping; and Jukka Komminaho, Uppmax, Uppsala for taking part in our survey and providing insights into how accounting problems are dealt with today.

10 References

[1]	The Global Grid Forum. http://www.ggf.org
[2]	Foster, I., Kesselman, C., Nick, J., and Tuecke, S., The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, Globus Project, 2002
[3]	Tuecke, S., Czajkowski, K., Foster, I., Frey, J., Graham, S., Kesselman, C., Maquire, T., Sandholm, T., Snelling, D., and Vanderbilt, P. Open Grid Services Infrastructure (OGSI) Version 1.0, GGF, 2003.
[4]	Grid Resource Allocation Protocol. GGF Working Group. https://forge.gridforum.org/projects/graap-wg
[5]	Czajkowski, K., Dan, A., Rofrano, J. Tuecke, S., and Xu, M. Agreement-based Service Management (WS-Agreement) Version 0. GGF 2003. https://forge.gridforum.org/projects/graap-wg/document/WS-Agreement_specification/en/1/WS-Agreement_specification.doc
[6]	Job Description Language. GGF Working Group. http://www.epcc.ed.ac.uk/~ali/WORK/GGF/JSDL-WG/
[7]	OGSA Resource Usage Service. GGF Working Group. https://forge.gridforum.org/projects/rus-wg
[8]	Usage Record. GGF Working Group. https://forge.gridforum.org/projects/ur-wg/
[9]	Grid Economic Services Architecture. GGF Working Group. https://forge.gridforum.org/projects/gesa-wg
[10]	Gagliardi, F. et al. GRIDSTART Project – IST Grid Projects Inventory and Roadmap. GRIDSTART-IR-D2.21.2-V0.5. EU 2003. http://www.gridstart.org/GRIDSTART-IR-D2.2.1.2-V0.5.doc
[11]	AVO, http://www.euro-vo.org
[12]	EGSO, http://www.mssl.ucl.ac.uk/grid/egso/egso_top.html
[13]	FlowGrid (FLOW simulations on-demand using GRID computing), http://www.unizar.es/flowgrid/
[14]	Eurogrid, http://www.eurogrid.org
[15]	GRIA, http://www.gria.org
[16]	GEMSS (Grid-Enabled Medical Simulation Services), http://www.ccrl-nece.de/gemss/
[17]	GRASP (GRid based Application Service Provision) http://eu-grasp.net/
[18]	Guarise, A., Piro, R., and Werbrouck, A. DataGrid Accounting System – Architecture –

	v1.0. DataGrid-01-TED-0126-1_0. EU DataGrid 2003. http://server11.infn.it/workload-grid/docs/DataGrid-01-TED-0126-1_0.pdf
[19]	Abramson, D., Giddy, J. and Kotler, L. High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?, International Parallel and Distributed Processing Symposium (IPDPS), pp 520- 528, Cancun, Mexico, May 2000.
[20]	Hazelwood, V., Bean, R., and Yoshimoto, K. SNUPI: A Grid Accounting and Performance System Employing Portal Services and RDBMS Back-end. Linux Clusters: The HPC Revolution, Urbana/Champaign, USA, June 2001.
[21]	Thigpen, W. et al. Distributed Accounting on the Grid. http://www.psc.edu/~lfm/dawg_vis4.PDF
[22]	Gridmap patch for Globus. http://www.gridpp.ac.uk/gridmapdir/
[23]	Keahey, K., Ripeanu, M., and Doering, K. Dynamic Creation and Management of Runtime Environments in the Grid. To appear in GGF9 Proc, Chicago, IL, USA October 2003.
[24]	EU DataGrid: Virtual Organization Membership Service. http://hep-project-grid-scg.web.cern.ch/hep-project-grid-scg/voms.html
[25]	Globus: Community Authorization Service. http://www.globus.org/security/CAS/

Appendix A: Bank Service Interface

Since a job submission might take quite some time it would not be feasible to keep a connection to the bank for the duration of the job. Rather, a new connection for each request must be made. Such a request would have the following general form:

1. Mutual authentication and authorization
2. User (or entity acting on user's behalf) makes a request
3. Bank replies

The bank service will at least need the following operations: (Note that these bank operations assume an in-blanco signing approach to service agreements. For a contract signing approach, some operations would need to be modified/added/removed.)

Operation: isDNAccountHolder
Purpose: Ask bank service whether a specific user has a bank account (for a specified VO-project).
Input: DN, VO, project
Output: true/false

Operation: acquireHold
Purpose: Acquire a time-limited reservation on an amount of the specified project account.
Input: DN, VO, project, amount, releaseTime
Output: hold-id (which can be used by the acquirer later to refer to hold) if hold could be acquired, otherwise an error.

Operation: extendHold

Purpose: Extend the (soft-state) timeout on the hold.
Input: DN, VO, project, hold-id, newReleaseTime
Output: OK/error

Operation: releaseHold
Purpose: Release the hold on user's account. This invalidates the hold, making it impossible to withdraw any amount from the user account using this hold.
Input: DN, VO, project, hold-id
Output: OK/error

Operation: validateHold
Purpose: Ask bank service whether a specific hold is valid.
Input: DN, VO, project, hold-id
Output: VALID/INVALID
Comment: Used in case user/broker-acquired holds should be forwardable to resource. Then resource will need a way of ensuring that the received hold on the user's account is authentic.

Operation: withdraw
Purpose: Withdraw specified amount from user's account. Credits are withdrawn from the pool of funds reserved when acquiring the specified hold. Any residual amount is released from the hold and returned to the user's account.
Input: DN, VO, project, hold-id, UR
Output: OK/error

Operation: getTransactions
Purpose: Get a list of user's completed transactions
Input: DN, VO, project, [from-date, to-date]
Output: List of transactions or error.

Operation: getBalance
Purpose: Get user's current account balance.
Input: DN, VO, project
Output: Account balance/error

Operation: createAccount
Purpose: Create a bank account for a user.
Input: DN, VO, project, initial-amount
Output: OK/error
Comment: Only available to privileged users.

Operation: deleteAccount
Purpose: Remove a user's bank account
Input: DN, VO, project
Output: OK/error
Comment: Only available to privileged users.

Operation: getAccountHolders
Purpose: Get a list of the all account holders in bank.
Input: [VO-set, project-set]
Output: List of user accounts/error
Comment: Only available to privileged users.

Operation: updateAccount
Purpose: Update available funds on an account
Input: DN, VO, project, amount
Output: OK/error
Comment: Only available to privileged users.