Dynamic resource allocation and accounting in VOs

Workpackage:	5 Grid Dynamics	
Author(s):	Sven van den Berghe Mike Surridge, Thomas Leonard	Fujitsu Laboratories of Europe IT Innovation
Authorized by	Mike Surridge	IT Innovation
Doc Ref:	P5.4.3	
Reviewer	Barbara Cantalupo	DATAMAT
Dissemination Level	Public	

Date	Author	Comments	Version	Status
2005-08-03	SvdB	First partial draft	0.1	Draft
2005-08-27	MS	Added IT Innovation contribs on GRIA, plus background section.	0.2	Draft
2005-09-02	MS	Added conclusions of meeting with Sven of 2005-09-02.	0.3	Draft
2005-09-10	SvdB	Filled in section 5, revised	0.4	Draft
2005-09-11	MS	Added Thomas contribution and filled in section 6.	0.5	Draft
2005-09-14	SvdB	Executive summary, cleaned for first pass at internal QA	0.6	Draft
2005-09-16	SvdB	Changes after review	1.0	Accepted

Executive Summary

The starting points for the work reported here were the early findings of WP5, concerning the different styles of VOs supported by Grids, and the subsequent development of business Grid concepts including the idea of bipartite SLAs governing interactions.

Our approach to the problem had three stages. First, the processes of resource and user federation were analysed from a NextGRID architectural perspective to identify a simple set of architectural principles or primitives that could define how federation should be achieved in NextGRID. Next, an analysis was made of federation mechanisms in existing Grid middleware: Unicore, GRIA and Globus. This was used to check that the proposed NextGRID federation primitives are general enough to model a wide range of existing Grid-based VO paradigms. Finally accountability was analysed in the context of the NextGRID primitives and SLAs. This work was performed in order to derive a NextGRID model of accountability that can be included as terms in bipartite SLAs.

The conclusions of this work are as follows.

1. We propose that there are 3 federation primitives that should be supported by the NextGRID architecture.
2. Delegation is an overloaded term, which has caused some problems; this can be corrected by a precise use of the term.
3. We have analysed three existing Grids (Globus, Unicore and GRIA), and conclude that the three federation primitives are more than sufficient to implement their federation models. Indeed Globus and Unicore exhibit only (dynamic) orchestration, while GRIA also has consumer federation but only static orchestration. None of the systems explicitly recognise encapsulation.
4. We have also examined how the federation models from these three current-generation Grids could be expressed through SLAs as defined by NextGRID, and found that it is feasible to do this, and to automate orchestration actions governed by them.
5. We note that all these SLAs should have clauses to prevent untrustworthy actions of delegates in service orchestration, which in these current Grids could not easily be detected by the delegator.
6. We have proposed a set of requirements for a “statement of accountability” that should be a part of every NextGRID SLA.
7. We have analysed and documented the functionality requirements for a generic accounting service.

We recommend that WP3 take account of the different types of SLA clauses we found were needed to represent behaviour and accountability in typical current Grid systems, and also the need to have SLAs that can refer to other SLAs in the same or related contexts.

We recommend that WP4 and WP5 should investigate how a more flexible dynamic security implementation could be used to reduce business dependency and thus simplify the number of non-automated “business trust” terms needed in SLAs.

We recommend that WP5 should develop a generic accounting service, based on the NextGRID dynamic security core, to form part of the common infrastructure and provide a standard way to manage accountability and create billing records in NextGRID.

The accounting service should be used in a series of experiments that examine the ways that statements of accountability can be formed and the terms that they should carry

Table of Contents

1 Introduction

This report constitutes Project Output P5.4.3 “Dynamic resource allocation and accounting in VOs” under EC IST Project 511563 “The Next Generation Grid” (NextGRID) [1].

The starting points for the work reported here were, firstly, the early findings of NextGRID WP5, concerning the different styles of Virtual Organisations (VOs) supported by Grids, and, secondly, the subsequent development of business Grid concepts - leading NextGRID to move towards a B2B model of participant interactions governed by bipartite Service Level Agreements (SLAs). The background and significance of these developments for the work reported here and the approach subsequently taken are discussed in Section 2.

The approach taken was to start by analysing Grid federation mechanisms in light of the NexGRID architectural approach. This is discussed in Section 3, where three primitive federation mechanisms are identified. This Section also discusses the relationship between federation mechanisms (and service provision in general) and “delegation”, a term normally associated with authorisation mechanisms in Grid circles, but which has a much broader meaning in English. This broader meaning is more widely applicable to business and organisational characteristics.

Having established a base set of federation mechanisms important to NextGRID, a selection of current Grid systems is then analysed in Section 4: Globus, Unicore and GRIA. All can be described in terms of a restricted subset of the NextGRID federation primitives. They do have relatively distinctive models of accountability, though none make this explicit in the form of SLAs.

Section 5 provides an analysis of the findings from this study, and describes a possible general (NextGRID) model of accountability that could be incorporated in bipartite SLAs. This suggests that it should be possible to define a generic NextGRID accounting facility, and a specification for a service to implement this facility is presented in Section 6

Finally, Section 7 provides a summary of the conclusions of the work, plus recommendations for development of the proposed accounting service in the next phase of NextGRID and its use in further experiments.

2 Background and Objectives

2.1 NextGRID and the Virtual Organisation

The overall objective of the NextGRID project, as described in its Description of Work [1], is to define the architecture for the next generation in Grid middleware. The work described in this report was carried out in support of this aim, by investigating issues related to resource allocation and accounting, *in the context* of the picture emerging from the architectural analysis based on similar investigations across the entire project.

When the original Description of Work was drafted, one of the areas of interest to the project was the manner in which Virtual Organisations (VO) would operate on the Grid. At that time, it was widely assumed that VO must exist between resource owners, and that users would access these resources as though they comprised a single resource pool controlled and accounted for by the VO.

In practice, the first six months of work in NextGRID WP3 (Business and Operational Issues) and WP5 (Grid Dynamics) showed that these assumptions, while often true in Grid operations, are in fact too limited. As described in Section 3 of [2], two types of virtual organisation were identified early in the project, only one of which (initially characterised by the term "big VO") involves direct collaboration and sharing between resource providers. The second type of VO emerged much more recently from work in Web Service B2B Grids such as GRIA [3]and GEMSS [4], and was initially called a "fast VO" model. In a fast VO, the resource providers do not collaborate directly, instead they sell services to resource consumers. Resource consumers may choose to federate services from different providers and may use them to collaborate with each other in a kind of decentralised VO.

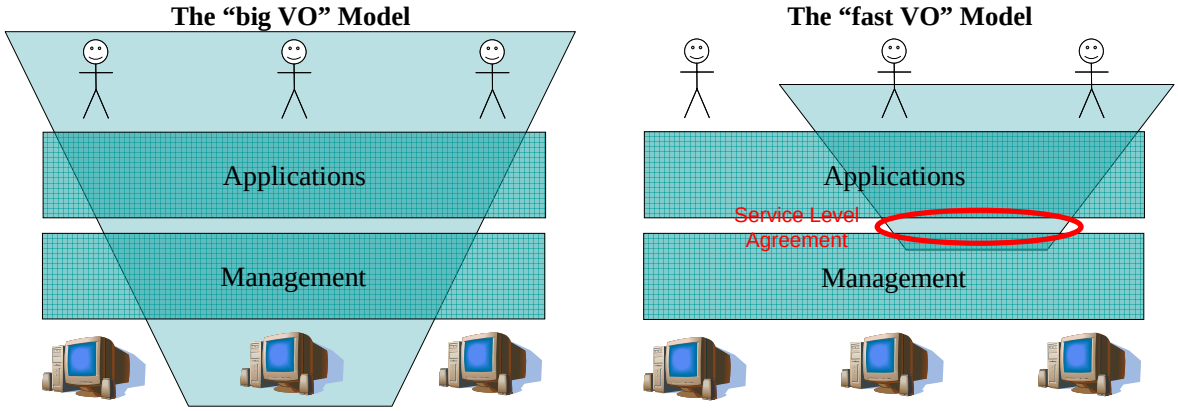

Figure 1. Initial classification of Virtual Organisation (VO) models

It quickly became apparent that these two VO models were actually representatives from a much larger range of possibilities, capable of supporting a much wider range of collaborative business models. In practice, the nature of the VO in a commercial Grid will be determined by the Service Level Agreements (SLAs) between the service providers and consumers. These should dictate (among other things) when and how services can be federated, and which actors can initiate or control the resulting multi-enterprise collaboration.

2.2 Specific Objectives

This was just one of several findings in the first 6 months of the project that led NextGRID to focus on the role and nature of bipartite Service Level Agreements. The current architectural hypothesis in NextGRID is that the interactions between any two organisations participating in a Grid should be covered by such an agreement.

If this principle is valid, then bipartite, commercial SLAs should play the fundamental role, and allow a wide variety of different types of VO to be realised. To enable this, the mechanisms whereby resources are federated and managed in a VO (including resource allocation and accounting) should be described in the SLA. In this context, the goals for the work described in this report shifted from a study of resource allocation and accounting mechanisms in Virtual Organisations, to a study of how such issues can be addressed in a general way in SLAs.

2.3 Approach

There is almost no prior work on this approach to Grid-based construction and operation of Virtual Organisations and Enterprises. Both GRIA and GEMSS introduced a form of SLA in the context of B2B Web Service Grids, and in GEMSS this evolved to conformity with the WSLA specification proposed by IBM (now at Version 1.0 [5]). However, in both projects the "fast VO" resource virtualisation model was implicit in the middleware implementation, just as the "big VO" model was implicit in earlier Grid middleware such as Globus. Neither GRIA

nor GEMSS took the further step of explicitly stating their federation mechanisms in their SLAs, let alone allowing the operation and business consequences of these mechanisms to be negotiable.

Our approach was therefore to attack the problem in three stages.

1. First, the processes of resource and user federation were analysed from a NextGRID architectural perspective. The goal was to identify a simple set of architectural principles or primitives that could define how federation should be achieved in NextGRID.
2. Next, an analysis was made of federation mechanisms in existing Grid middleware including Unicore, GRIA and Globus (chosen because they exhibit a range of characteristics from “big” and “fast” VO). This was used to check that the proposed NextGRID federation primitives are at least general enough to model a wide range of existing Grid-based VO paradigms.
3. Finally, accountability, and particularly billing, was analysed in the context of the NextGRID primitives and SLAs. This work was performed in order to derive a NextGRID model of accountability that can be include as terms in bipartite SLAs. This provided a starting point for the design of a NextGRID accounting service that will be used in experiments to validate the accountability model and its inclusion in SLAs.

In principle, almost any aspect of resource federation could become subject to a negotiated SLA between service providers and consumers, but to conduct an analysis under such general assumptions was not feasible. It is expected that the existing analysis will be extended later in the project to address other aspects as and when they become important in NextGRID. For this analysis, we chose to focus on two specific business aspects of user and resource federation: namely accountability and billing.

In this document, Accounting is taken to mean the complete process of acquiring records of any activity using resources on a NextGRID, storing these records, and making them available for further analysis. Furthermore, Accountability is taken to mean that all actions can be traced to a user who is prepared to be held responsible for the actions. Note that in the context of the federation primitives discussed above, the accountable user may not be the user who initiated the actions.

The particular aspect of Accounting that is the focus of this analysis is Billing. In a commercial Grid any use of resources will have to be paid for, either directly or indirectly and this charge must be applied to the correct, or accountable, user. As with the definition of accountability above, the entity charged for using resources in NextGRID may not be the entity that actually uses the resources; there may be more than one identity involved in a chargeable interaction[1]. Thus, Billing is related to accountability for actions and we believe that the principles discussed here are also applicable to the other areas of Accounting as well as to broader areas of Grid Dynamics.

.

[1] The process of authentication of these entities is not considered in the scope of this analysis; we assume that this can be done within a NextGRID and made available to the accounting modules

3 Architectural Analysis

3.1 NextGRID Architectural Principles

The NextGRID architecture changed significantly during the course of this work, in part stimulated by the early findings of this work. At present, the NextGRID consortium expects [6] that the architecture should obey three fundamental principles:

- the interactions between any two actors participating in a Grid should be governed by a bipartite Service Level Agreement;
- applications (and other multi-lateral federations) should be constructed through the composition of services architecturally (self) similar
- there should be a minimal set of services, properties and behaviour available in a NextGRID, which characterise NextGRID and provide a base platform on which developers can depend.

The first of these principles was driven by the need to address commercial Grid scenarios. The second was driven by a desire for architectural simplicity, but also by the analysis of federation mechanisms described below.

3.2 Atomic Service Provision and Context

Before considering federation mechanisms, we must first specify the nature of atomic interactions involving a single client consumer invoking a service from a single service provider. The interaction between the service provider and consumer is governed by a Service Level Agreement, which is assumed to be negotiated prior to the service invocation process. The SLA, along with context information on each side determines how each side participates in the interaction. Note that while the SLA is shared between the two parties, the context in which they interpret the SLA includes private information that is not shared. This situation is shown in the figure below, where the client process is Consumer (A) and the server process is the Service (B).

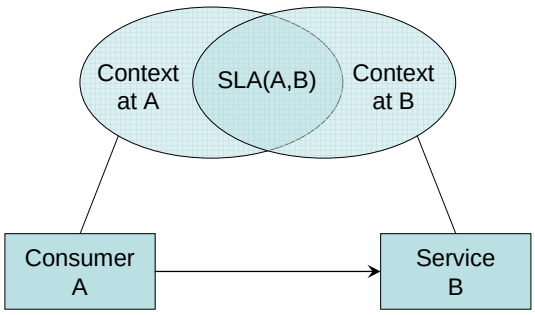

Figure 2. Atomic Service Provision

In this interaction the authority of the consumer (A) to perform the invocation is determined by the service (B) directly from the negotiated SLA, along with the (possibly private) context at B. This private context may include the history of all previous interactions with B (whether or not involving A), and undisclosed policies that may be related to this (e.g. a blacklist of certification authorities used previously by malicious crackers), etc.

Note that some things normally have to be exchanged between A and B before an SLA can be agreed. At this stage, it is left open whether these formally constitute an extra (or prior) shared context that is outside any SLA, a co-incidence between parts of formally private context on

each side, or an implicit SLA whose terms may be nothing but acceptance of the same protocols as a starting point for subsequent exchanges.

3.3 Federation Primitives

3.3.1 Encapsulation

Encapsulation is the act of using one service to provide another service, with no direct interactions between the consumer of the first service (A) and the provider of the second service (C):

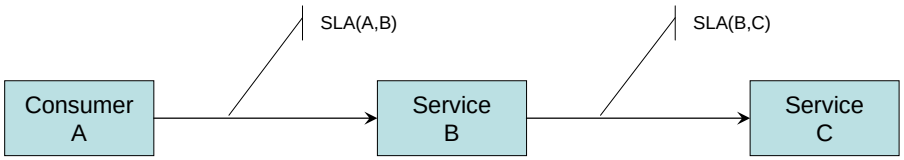

Figure 3. Service Encapsulation

The effect of this is to federate resources (from B and C) and make them available to A through B.

Encapsulation can begin at any time during the interaction. The intermediary (B) may already be providing service to consumer (A) and then decide to outsource some of the work to an encapsulated service (C). Or the intermediary may have established their relationship with the encapsulated service (C) prior to offering services to the end-consumer (A). However, in all cases the decision to invoke the encapsulated service is made by the intermediary service (B) and not by the first consumer (A).

In Figure 3, SLA(B,C) clearly forms part of the context in which B provides service to A. However, there is no necessity for B to share information from C with A. Moreover, the terms of the SLA between B and A need not reflect those between B and C, or vice versa. In many cases, the actions of B with respect to one party, including their position during SLA negotiation, will be influenced by the terms of their interaction with the other party. However, this should be left as a business decision for B, and cannot be assumed derivable in an automated way from the architecture for encapsulation.

3.3.2 Orchestration

Orchestration is the act of asking two services to interact directly on behalf of a common consumer. The basic scenario is for a consumer (A), through their interactions with two service providers causes one of them (B) to invoke services at the other (C):

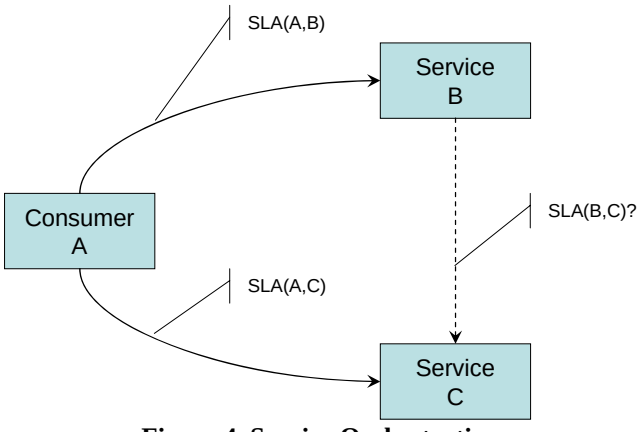

Figure 4. Service Orchestration

Like encapsulation, the effect of orchestration is to federate resources (from service providers B and C) and make them available to the consumer (A). However, in orchestration the consumer makes the decision to initiate the interaction between B and C, raising the possibility that the SLA covering this interaction may refer to the common third party A. In encapsulation, each SLA is self-contained and separately negotiated, but in orchestration it is feasible to generate the SLA between B and C automatically based on their interactions with A. In this case, the procedure if not the terms of SLA(B,C) must be foreseen and defined through their SLAs with A.

3.3.3 Consumer collaboration

The last federation primitive is where the consumer (A) of a service (B) shares access to this service with another consumer (D):

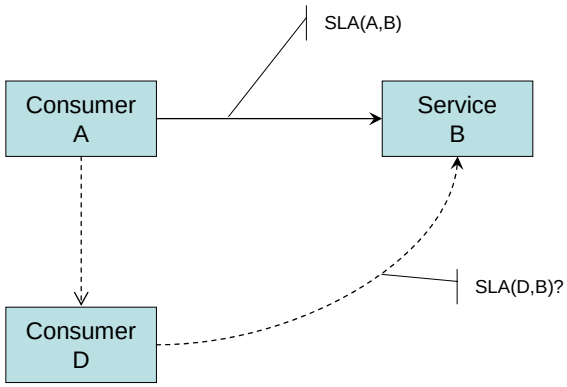

Figure 5. Consumer collaboration

This scenario doesn't federate resources, but it does federate consumers (users), and so enables them to collaborate. It is thus fundamental to the development of Virtual Organisations uniting users of a set of services.

The key difference between the Collaboration and Orchestration scenarios, is that users rarely negotiate formal SLAs, so that the computational infrastructure must infer the SLA between D and B using only one other SLA (between A and B) as its initial context. As with orchestration, all three parties are mutually aware in this scenario, and it is possible for SLA(D,B) to be inferred through some automatic procedure. Even if this is not the case, and D has to negotiate with B before gaining shared access to resources, the possibility of consumers using the service collaboratively, and the procedures and restrictions for this collaboration should clearly always be foreseen in the terms of SLA(A,B),

3.4 Delegation: Truth and Fiction

The English definition [7]for the word "delegation" is:

> *The action of delegating or fact of being delegated; appointment or commission of a person as a delegate or representative; the entrusting of authority to a delegate.*

This concept in its most general sense is relevant even in atomic service provision; whenever a consumer asks a service to perform a task for them, they are in fact delegating the conduct of the task to the service. In this sense, encapsulation introduces nothing new – service B in Figure 3 is simply deciding for itself to engage a third party to implement the task requested by A. Orchestration may involve something more, since access from B to C in Figure 4 may

depend on authority held by A, which must in some way be entrusted to B to enable B to fulfil the orchestrating instruction. Similarly, consumer federation (Figure 5) involves A entrusting D with access rights at B. In all cases, "delegation" is the act of asking another to undertake a task, in which the entrusting of authority to the delegate may or may not be needed.

In the Grid world, the English definition of the word has been forgotten, and the term is now used to express the narrower sub-concept of entrusting authority to a delegate. The best-known implementation of this narrower form of delegation in current Grids is the use of GSI proxies [8] to implement service orchestration in Globus. In the GSI mechanism, the delegate (B) acquires the identity of the delegator (A), allowing it to access other services (e.g. C) using the same rights held by the delegator. However, even this form of delegation can be carried out by other means – Unicore uses the notion of Explicit Trust Delegation where the intermediary (B) is identified to the Service (C) but the interaction carries a token stating that A has delegated to B the ability to carry out (certain) actions on its behalf.

In NextGRID, any of the federation mechanism: service encapsulation, orchestration or consumer federation *MAY* involve the entrusting of authority to a delegate by a delegator. This is one of the parameters of the interactions within a federation that should be defined through an SLA. It is important to keep this (and the different meanings of delegation, both narrow and broad) in mind when examining existing Grid implementations.

4 Current Grids

4.1 Globus

4.1.1 Service provision model

Globus [9] is a file-compute Grid infrastructure designed to support remote access to resources shared within a Virtual Organisation of resource providers and their users. A range of other services are available that are compatible with Globus, including data services (e.g. the WSRF version of OGSA-DAI [10] from EPCC and IBM) and services for implementing traditional "Big" Virtual Organisations (e.g. the Community Authorisation Service or CAS [11]). Here, we focus on job execution and consider other services only where relevant to issues to accountability or billing.

Jobs to be executed in Globus are defined using a Globus-specific XML schema, in which the user's commands are specified along with any remote data movements (staging directives). This specification is then sent to a Globus GRAM or WS-GRAM service, which will determine the user's Grid identity based on either transport or message level authentication, and attempt to map this Grid identity onto a local identity. If this is successful, and any other authorisation policies are satisfied, then the job will be executed using the local identity mapped to the user's grid identity.

Where Globus jobs involve access to other services (e.g. to conduct data movements or to run sub-jobs), the service provider invokes these third-party services using a GSI proxy credential asserting the Grid identity of the original user. Thus the service "impersonates" the user, and the third-party service authenticates the request and applies authorisation policies as though it was being made by the original user. This mechanism is fundamental to the integration of Globus with data services that also support it, including OGSA-DAI and SRB [12].

Globus jobs can also be sent via an intermediate service such schedulers and brokers (e.g. Plaform CSF [13]), which acts as a broker between the user and the individual service

providers. These intermediaries also use GSI proxies, so that when the job eventually arrives at the execution node, the request to run it can still be authenticated as coming from the original user. Indeed, the intermediary can then propagate the GSI identity to the execution node, so the impersonation process can still be used for data access, as above.

4.1.2 Relationship to NextGRID primitives

At first sight, it appears that the GSI proxy mechanism is flexible enough to support both orchestration (when a job service reads or writes data to a third-party data service) and encapsulation (when a broker service selects an execution node and runs a job on behalf of the user).

In fact, it is clear that GSI is only really applicable to orchestration. All the services (even in the brokered execution scenario) authenticate service requests as coming from the original user, and execute them with the user's identity. The user must have a pre-existing relationship with each service involved, and the service handles their request in the context of that prior relationship. The user doesn't need to know which service will be used for any given job, so we should characterise this as dynamic orchestration, but it is not encapsulation. In a true encapsulation, the first service would invoke encapsulated services using its identity, and the user would not need any prior relationship with the encapsulated service. This is not what happens in a GSI-based Grid such as Globus.

It is also clear that Globus does not directly support consumer collaboration, in that there are no generic mechanisms for sharing services between *different* consumers. GSI can be used to allow one user to impersonate another and so gain access to a service, but this is not the intended purpose of the GSI mechanism, even though the term "delegation" if interpreted literally might suggest it. Some data services that can be used with Globus do support consumer federation (e.g. the issuing of tickets to enable access to data sets in SRB), but this feature is distinct from the core Globus federation model based on GSI.

4.1.3 Accountability and billing

The GSI mechanism for resource federation is both a strength and a weakness of Globus. It allows fully dynamic orchestration, but it also assumes that all the services involved can be trusted implicitly to manage and use GSI proxies only on behalf of the user. This makes it a risky choice in a commercial Grid, especially if open to anyone who can pay for services.

In NextGRID, such "unconstrained" freedoms should not be permitted, except when specified through SLAs between service providers and their users. To implement Globus models in NextGRID, the SLA would need to specify the following terms:

1. The service will recognise requests as coming from the user if authenticated as such using a GSI proxy.
2. The service will process the request if it recognises the user as being authorised to use its service.
3. The user is responsible for all consequences of processing such a request.
4. The user will supply the service with a GSI proxy to be used if the requested action requires access to any other services.
5. The service will provide further GSI proxies to these other services on behalf of the user, if required by them to do so.
6. The service will protect this GSI proxy, and use it only to execute actions requested by the user.

Globus doesn't support an explicit billing process, but the model is clearly consistent with the idea that the user will be billed for any services where payment is required.

With the SLA clauses indicated above, accountability in Figure 4 is determined by the fact that the service provider makes no distinction between a user and their GSI-authenticated delegates. This means the orchestrated interaction between B and C is interpreted as coming from A, and the applicable SLA as seen by service C is just SLA(A,C).

When spelt out in an SLA, the risks inherent in the combination of clauses 1 and 5 become more apparent. Of course, this doesn't mean the Globus model is not useful – the benefits of using GSI for dynamic orchestration may outweigh the potential risks in many cases. It would be possible to introduce some additional risk management by adding conditions for the creation of GSI proxies in clause 5, and use these to limit their acceptability in clause 1.

It is also clear that terms equivalent to the above could easily be implemented without using GSI – the risks are not specific to GSI, although when using GSI it is harder to avoid them.

4.2 Unicore

4.2.1 Basic service provision model

Unicore [14] is a file-compute Grid infrastructure designed to support job execution on shared resources.

Actions to be performed on a Unicore Grid are described in a Java object called an Abstract Job Object (AJO). An AJO is executed at one site, multi-site jobs are created by including sub-AJOs within a parent AJO. An end user sends an AJO to a Unicore server for execution. Sub-AJOs in multi-site jobs are transferred from server to server. The process of sending an AJO from end-user to server or from server to server is known as consigning.

Including sub-AJOs in a job means that the user can delegate to the primary execution site the right to execute part of the work on their behalf at another site. Basic Unicore recognises two roles in the delegation process:

- The endorser is the end-user to whom the AJO belongs (i.e. who created the AJO and under whose authority the AJO will be executed). Endorsement of the AJO is shown by transferring a digital signature of the AJO with the AJO. This digital signature is created using the private key bound to the endorser certificate. A valid digital signature of an AJO is taken to be authorisation by the end-user for the actions described in the AJO to be performed by the server.
- The consignor is the agent that transferred the AJO to the server; this is either the end-user or another Unicore server (passing on a previously endorsed sub-AJOs). The consignor certificate is extracted from the socket level connection used to transfer the AJO to the server.

When an end-user consigns an AJO, the end-user's certificate is used for both the Consignor and Endorser roles. However, when a sub-AJO is consigned between two servers, the end-user's certificate has already been used to endorse the sub-AJO but the server's certificate is used to establish the connection and thus consign the sub-AJO i.e. the server certificate takes the Consignor role.

An end-user is allowed to endorse and consign AJOs; a server can only consign AJOs. This means that server processes cannot create jobs. A server can only consign jobs that have been endorsed by end-users. This allows servers to consign sub-jobs, but these sub-jobs will be executed with the delegated identity of the Endorser. An end-user may only consign jobs that she has endorsed. These rules mean that sites running Unicore do not need to trust other Unicore sites in order to receive sub-AJOs from them. All AJOs they receive must have been created by end-users and can't be modified by intermediary servers. There is no need in Unicore to contact any other parties to create a delegated job, endorsing is an activity carried out entirely on the client side.

An essential part of the Unicore approach is the separation of authentication from authorisation. Certificates are used solely for authentication and carry no authorisation information. Authorisation of actions is contained within the signed AJO and authorisation of users within sites is performed by the server using site local rules.

The Basic Unicore model is static. The end-user describes the steps that a job will take, including the resources that are required to execute each step, before the job is submitted. This description is signed and the Unicore servers execute this description if it has not been changed since signing. Although the description allows some flexibility, such as conditional execution of parts of the job and a generic request to "run anywhere suitable", there is not the flexibility to adapt to changing circumstances that are required by NextGRID applications.

4.2.2 Extended service provision model

As mentioned above, when AJOs are consigned between end-user and server, only two roles are involved: consignor and endorser. In this Basic Unicore model, the Endorser is used for two purposes, to authorize the actions (by signing the AJO) and also to identify on whose behalf the actions are to be performed. This means that a set of actions can only be created by the person who wishes to execute them i.e. a user cannot delegate the ability to create a set of actions. The Explicit Trust Delegation [15] extension to Unicore adds a new role to the model in order to separate these two purposes. The User role identifies the end-user on whose behalf the action is to be performed. The Endorser role is retained but now only authorizes the actions (still by signing the AJO).

Unicore servers supporting Explicit Trust Delegation implement the following set of rules to define the behaviour of each of the three roles:

- Only the User role is mapped to a local identity and has tasks executed on their behalf.
- Only end-users can take on the User role.
- End-users and Unicore agents (e.g. servers, portals, resource brokers, schedulers, etc.) may take on the Consignor role.
- Unicore agents may take on the Endorser role to create actions on behalf of an end-user.
- Any Endorser, which is not the same as the User, must be explicitly entered into the local authorisation database as a trusted agent allowed to endorse end-user's jobs.

During processing of an AJO at a server, the server authenticates the Consignor and Endorser, ensures that the User is authorized access to the site, and verifies that, if the Endorser is not the User, the Endorser is authorized to endorse on behalf of the User. All this information is coded explicitly into the authorisation database maintained at each site.

The introduction of the User role into the Unicore model allows dynamic delegation since a set of actions can be created by a third party on behalf on an end-user without the end-user needing to explicitly authorise the actions.

4.2.3 Relationship to NextGRID primitives

In Unicore, two forms of "delegation" are supported:

- a basic version in which users must sign sub-jobs so they can be transferred by one service to another;
- an extended version in which a service is designated as an endorser, allowing this service to construct its own sub-jobs and transfer them on the user's behalf to another service.

These models are examples of static (predetermined) and dynamic (service mediated) service orchestration.

Unicore does not explicitly support service encapsulation. Clearly, it is possible for a Unicore service provider to implement a supported application using services from third parties. However, the service provider would have to actively reassert its own identity as a user of services (rather than as a consignor of user's requests) when outsourcing part of the work to a third party.

Unicore also does not support consumer collaboration. AJOs are only accepted from users if the user consigning the AJO endorses them and so AJOs cannot be shared between users. This is not surprising, given that Unicore is a job-centric Grid – consumer collaboration is normally associated with data grids, with access to data services being the resource that is shared among the consumers.

4.2.4 Accountability and billing

In both Unicore orchestration scenarios, whether endorsed by the user or by a designated service delegate, when a job is executed at a service provider site, the service provider treats the user as being accountable. It would make sense for resource consumption by each job to be billed to the user who initiated that job.

Given this, if the federation mechanisms in Unicore were to be used in NextGRID, the SLA would need to specify the following terms:

1. That the service will recognise and execute jobs endorsed by the user.
2. A list of third party consignors or a means of identifying them.
3. Whether the service will recognise and execute jobs endorsed by a third party service.
4. A list of third party endorsers or a means of identifying them.
5. That the users must be accountable for their use of the service, even for jobs consigned or endorsed by others subject to the above conditions.

Unicore is far less "trusting" than Globus, even if the extended service model with delegated endorsement is used. However, this would have to be controlled through the SLA, by adding some extra terms for services that may act as endorsers:

6. Whether the service will ever act as an authorised endorser for the user (it is useful to rule this out through the SLA if the user knows it will never be necessary).
7. That the service will not use its endorser role except to execute jobs on behalf of the user.

If NextGRID is to support Unicore mechanisms, aspects **Error! Reference source not found.** to 7 above should be supported in SLAs. In this event, the SLA applicable to an interaction between services for consignment or endorsement of a user job could be derived from their SLAs with the user. In effect, accountability in Figure 4 can be specified by copying or referring to the relevant terms from SLA(A,B) and SLA(A,C) in SLA(B,C).

There are no explicit billing facilities in the current version of Unicore, so the billing arrangements are "out of band", but must be established by each user by the time they agree the SLA in the "computational world". If billing services were added to Unicore, then it would be possible for users to specify a different billing mechanism for each job. In that case, the SLA would also need to cover

8. How the billing mechanism is defined for each job (e.g. through an end-point reference).
9. Whether this can be inserted by an endorser or consigner or only by the user.
10. What billing mechanisms are acceptable (e.g. only those of the service provider, or is there a list of third-parties such as banks that can be used).

Billing mechanisms are not needed to implement the Unicore models, and would involve some refinement of the existing Unicore delegation models as indicated by 8 to 10 above. This would be sufficient to define billing as well as accountability for orchestrated service-service interactions.

4.3 GRIA

4.3.1 Service provision model

GRIA [16] is a file-compute Grid infrastructure based on Web Services, but designed to support well-defined business processes for commercial service provision, and collaboration between different users of the same services.

The basic job execution model in GRIA involves a combination of its data and job execution services. Input data sets must first be uploaded to data services, at locations specified by URI assigned by the data service providers. Jobs are started by sending a request to a job service for the desired application, quoting the URI for the job instance (also assigned by the service provider) along with the URI for data sets to be retrieved as input, and for output data sets to be sent as output.

The assignment of URI by job and data service providers before jobs can be started or data transferred is a key aspect of the GRIA model. These URI provide a context for each request, and the service provider can associate (private) state with the URI, using the SessionFacade pattern [17]. This approach based on "hidden" context is a key aspect of GRIA, and is used in two ways:

- it allows the service provider to store an access control list (ACL) for operations on each job and data set (and other resources – see below), and update this dynamically based on consumer actions (a feature known as "process-based access control"); and
- it allows the service provider to implement processes for issuing job or data set URI that enforce business constraints on consumers of job and data services.

In the current version of GRIA (v4.2), two business-level services are included: a resource allocation service with which consumers must negotiate the use of computing and data transfer capacity, and an accounting service with which consumers must register a payment mechanism to receive a credit limit. These are also contextualised using URI with their own ACL to represent resource (capacity) allocations and billing accounts respectively.

Anyone can ask for an account URI, but the accounting service won't enable access (through the ACL) to billing functionality until the consumer's proposed payment mechanism has been checked and a credit limit assigned by the service operator. The only way to get a resource allocation URI is through the accounting service billing operations, so nobody can do this until they have a credit-worthy account. Initially, the ACL for a resource allocation URI enables access to the capacity negotiation functions only. Once the consumer has negotiated a capacity allocation, the functions to assign job and data URI are also enabled. The resources consumed by job execution and data transfer are eventually billed to the same account used to get the resource allocation URI. In this way, GRIA ensures that nobody can perform data transfers or processing without first providing an account where it can be billed, and negotiating an allocation of service provider resources.

The billing steps themselves are handled in three stages:

- the consumer's account is billed for capacity allocations at the point they are agreed with the resource allocation service;
- the consumer's account is refunded for any unused capacity at the point where this agreement expires or is terminated;
- the consumer's account is credited when they make payments to the service provider.

The first step fails if it would take the consumer's account over their credit limit. This places an upper limit on how much they can use GRIA services without paying. The service provider may choose to set the initial credit limit to zero, in which case the consumer will have to pay in advance of allocating any resource capacity from that service provider.

The percentage of the initial charge that is refunded for unused allocations is configurable by the GRIA service provider. However, GRIA is open with respect to the computational facilities used to store and process data – typically this is a cluster, but it is not assumed that it is dedicated or that it supports enforceable reservations. Because of this, the service provider rarely makes guarantees except that they will not allocate the same resources (whether or not they are really available) to more than one consumer. Consequently, the normal configuration is to refund 100% of the original charge for any unused capacity, and there is no additional penalty if the consumer failed to use the full capacity because it wasn't made available.

The GRIA services also provide management operations that allow other parties to be added to the ACL for specific service operations. This makes it possible to grant other consumers access to specific functionality, including:

- billing operations on an account URI, allowing colleagues to negotiate and use capacity charged to the same account;
- read and write access operations on a data URI, allowing colleagues (or other service providers) to access data.

GRIA does not use GSI proxies, and authority is always entrusted to delegates using these management operations. This means that GRIA users (providers and consumers of services)

can manage their trust relationships explicitly, but consumers must enable access in advance to any resources (URI) that a service or colleague needs to perform actions on their behalf. This trade-off (strong trust management but no dynamic delegation) was considered a good balance for an infrastructure designed to support industrial B2B applications.

4.3.2 Relationship to NextGRID primitives

The GRIA infrastructure uses two of the three proposed NextGRID federation primitives: consumer collaboration and (static) service orchestration.

Orchestration primitives are used extensively, just as in Globus and Unicore. The elevation of data storage to a first-class concept with its own supporting service means that whenever a user runs a job, they must orchestrate an interaction between the job service and data services for storing input and output. This is done by specifying the URI of the data sets (which may be stored at different sites), and leaving the job service to execute the necessary operations. The consumer must explicitly enable access by the job service provider to the data URI before starting the job. There are other examples of orchestration between services operated by the same provider, such as the interactions between job and data services and their resource allocation service to report their total resource usage. In these cases, the URI of a resource allocation is specified through the job and data URI creation process (following the Factory pattern), and there is no need to explicitly manage access rights as everything is handled by the same service provider.

It is important to note that unlike Globus and Unicore, all orchestration in GRIA 4.2 is static. In every service action that involves orchestration the other services to be contacted are specified directly or indirectly before the action is invoked.

Unlike Globus and Unicore, GRIA also makes use of consumer collaboration primitives. The purest example of this is the use of account management operations to enable access by colleagues to billing operations on an account URI. This feature is only used to support collaboration between groups of people working on a common project using (financial) resources assigned to that project. A second example is the management of access rights to data, which supports sharing of data between collaborating consumers, though this feature is also used in the orchestration of data transfers by job service providers.

Like Globus and Unicore, GRIA does not use encapsulation primitives, though services could be implemented that use this federation mechanism if necessary. Unlike Globus and Unicore, GRIA does not map Grid users onto local identities, so any service that uses another would in GRIA use their own identity by default. Thus identity is encapsulated by default, even if this is not currently exploited in GRIA.

4.3.3 Accountability and billing

Accountability and billing rules are explicitly designed into GRIA, but although these rules are quite sophisticated, they are not formulated in any SLA. The only SLA in GRIA relates to the allocation of resource capacity by a service provider against a consumer's resource allocation URI. Accountability in the wider sense is assumed, which would not be acceptable in NextGRID. To use GRIA accountability and billing mechanisms in NextGRID, clauses would need to be inserted into the SLA at different levels.

First, an SLA would be needed at the accounting service, specifying the following terms for each account URI managed by the service:

1. The consumer accepts accountability and agrees to pay for any action using resource allocations billed to the account.
2. The consumer may access billing functions only once the service provider has accepted their proposed payment mechanism.
3. The consumer may enable others to access billing functions on their account, but remains accountable for their actions under clause (1) above.
4. The service provider will prevent access to the account by anyone other than the consumer, unless permitted by clause (3) above.
5. The service provider will only allow their own resource allocation service to charge or refund to the account, and only for resource allocation URI billed to the account.
6. The service provider will credit the account whenever the consumer makes them a payment.

The resource allocation service would need the following SLA clauses for each resource allocation URI:

1. The service provider will prevent access to the resource allocation by anyone other than the consumer.
2. The consumer accepts that any resource allocations agreed will be charged to the account they specified for billing, and that accountability is governed by the SLA for that account.
3. If the consumer agrees a resource capacity allocation (amount and period) proposed by the service provider, then and only then may they access job and data set assignment functions to use this allocation.
4. When the consumer assigns resource limits to job and data sets, these resource limits must be within the total allocated capacity, and will be deducted from the capacity available for future allocations.
5. The service provider will only allow job and data sets assigned under this resource allocation to use or release resources from the allocation
6. If the consumer terminates the resource allocation, or the period over which it is valid expires, then any job or data sets assigned from it will be terminated by the service provider, and the price of any unused resources will be refunded to the billing account.

The data service would need the following SLA clauses for each data URI:

1. The consumer may transfer data to and from the service, or have the service retrieve or deliver data to and from other data URI, up to the transfer and storage limits specified when the data URI was created.
2. The service will refuse to transfer data to and from data URI unless the destination data service first confirms that the consumer also has the right to do so.
3. When the data URI is terminated, any unused capacity will be restored to the resource allocation from which it was created.
4. Accountability for any action on the data URI is specified by the SLA for this resource allocation.
5. The consumer may enable other consumers or service providers to access data stored at this URI, but still accountable under clause (4) above.
6. The service provider will prevent access to the data URI by anyone other than the consumer, unless permitted by clause (5) above.

Finally, the job service would require similar SLA clauses for each job URI:

1. The consumer may have the service retrieve or deliver data, and execute the specified application up to the data transfer, storage and CPU limits specified when the job URI was created.
2. The service will refuse to transfer data to and from data URI unless the destination data service first confirms that the consumer also has the right to do so.
3. When the job URI is terminated, any unused capacity will be restored to the resource allocation from which it was created.
4. Accountability for any action on the job URI is specified by the SLA for this resource allocation.
5. The service provider will prevent access to the job URI by anyone other than the consumer.

The only orchestrations in GRIA that span different service providers are those for transferring data between job and data services. Referring to Figure 4, and assuming service B is a GRIA job or data service, and C is another data service, the accountability terms of SLA(B,C) are actually determined by clause (4) from SLA(A,B) and SLA(A,C). This could clearly be automated in NextGRID, provided the SLA can be read and implemented by machines.

5 Analysis

5.1 Validation of NextGRID Federation Primitives

It is clear from the above investigation that the three federation primitives proposed in NextGRID are more than sufficient to describe current Grids:

- Globus and Unicore provide direct support only for (dynamic) orchestration of services;
- GRIA also supports consumer collaboration directly, but only has static orchestration.

The three primitives also appear to cover all the scenarios that might be implemented at the application level with any of these Grids. Clearly, it isn't necessary to handle all things the application developers may wish to do, but by supporting three federation primitives explicitly, NextGRID should provide far greater support to application developers and users.

5.2 Comparison of Accountability in Current Grids

There are two important aspects of managing accountability in Grids. The first is to determine who will be held accountable (or billed) for some actions and to pass this determination onto other parties in federated applications. The second is to control the risk that an accountable entity takes when using delegate services i.e. an accountable entity must trust the delegate service to perform just the work that it is requested to. The solutions to these aspects have implications for the degree and type of dynamism that is possible in a particular Grid infrastructure. In this section we analyse the accountability models of the three Grid systems with a view to determining which of these features are desirable in the NextGRID architecture.

The accountability for an action in Globus rests with a single entity – the user. This is traceable through the strong authentication within the GSI. Orchestrated services accept as tokens of accountability the impersonated identity of the user. Any client that identifies itself as a user (within the context of the VO) is treated as if it were the user. Thus any services within a VO can perform any action for which a user will be held accountable. Globus users trust services to perform only those actions that are required to execute the requested job. As brought out in the list of SLA terms required for Globus in Section 4.1.3, a Globus user also trusts a service to take care of a proxy certificate (i.e. by protecting its readability to prevent use of the proxy by

malicious services which, once they possess a proxy certificate, can impersonate the original user). Thus, the Globus model of accountability is best understood within the context of "big VOs" in which a number of resource owners come together to provide services to users in a relationship that is trustful. These relationships are long-term and have a large initial barrier to instantiating them. Part of this barrier is establishing the necessary trust. These big VOs are thus static, or at least slowly evolving. However, given the level of trust it is possible to relax the accountability restrictions to allow dynamic orchestration, in which the user does not pre-determine which services will be involved in executing their requests.

Accountability in Unicore also rests with a single user entity, which is traceable through the digital signatures that are attached to Job descriptions. Services involved in an orchestration accept the digital signatures as tokens of accountability and interpret these as authorisation to execute only the described job and not any arbitrary service. The level of trust that is placed in services that are asked to orchestrate is this much less than for Globus as it is limited to the execution of specified tasks. This in turn means that a VO in Unicore can be more dynamic than a Globus VO as it requires less "due diligence" to form and enrol new members. However, the execution of jobs is less dynamics since, to keep the signatures valid, a job need to be fully specified by the user before it is submitted for execution.

The enhanced version of Unicore (Section 4.2.2) adjusts this balance between trust in service providers and dynamism towards the Globus model by allowing services to create job descriptions for which the user will be accountable. However, this privilege is limited and must be explicitly granted for it to be accepted by other services.

The design of GRIA was influenced from the outset by business processes and so its model of accountability is more sophisticated and more explicit than Globus or Unicore. GRIA allows service orchestrations through collaboration and so, at first sight, there are two entities that could be accountable for an action; the consumer who created the accounting service or any of his delegates who share use of the accounting service to bill for resources and execute jobs. In GRIA, the service provider will only accept accounting service owner as accountable as the manipulation of access rights cannot constitute a change of accountability and so, as with the other two systems, there is only one accountable entity, though it is not always the user who initiated the process for which this entity is accountable. A Grid formed within GRIA has more dynamic membership than Unicore or Globus as the participating service providers are chosen by the consumer. However, orchestration of services within a job is static.

All three current systems have implicit assumptions on the use of authority entrusted to services acting as delegates of their consumers. The presence of such clauses in the (unwritten) SLA for these current-generation Grids is a source of concern in business applications. Making them explicit in NextGRID SLA should help business users understand and manage risks.

5.3 Generalisation of Accountability in NextGRID

Every NextGRID SLA should contain a statement of accountability that allows explicit determination of which entity is accountable for the actions to be carried within the context of the SLA (including billing). This statement will have two major sections:

1) Identifier
2) Scope

The identifier identifies the entity that will be accountable, explicitly by means of a X509 certificate or through naming a role (in more general accountability statements).

The scope restricts the applicability of the statement. The following are areas of restriction:

1) **Actions**. A statement of accountability should not be general. It should be restricted to a limited set of actions (e.g. the signed AJO in Unicore).
2) **Time**. A statement of accountability can be limited to a certain time period (as is currently done with GSI proxies). This could be the lifetime of the service or, for statements that are transferred (used in negotiating other SLAs) as defined amount of time.
3) **Services**. A statement can be limited to be applicable to a restricted number of services e.g. only the services that are part of a particular VO or only the services in a list etc
4) **Transferability**. Statements of accountability can be restricted to a single SLA or allowed to be transferred to other SLAs (one area of investigation is the desirability of restricting the number of generations of transfer allowed).

The variations of restrictions allowed in the scope will determine the characteristics of the VO – in the big to fast range.

SLAs can refer to accountability statements rather than include them. These references can be to SLAs created for other interactions e.g. in a collaboration (Figure 5) SLA(D,B) would use a reference to the accountability statement in SLA(A,B) as its accountability statement. These references can also be to standard statements that, for example, apply to all interactions in the context of a particular VO.

6 NextGRID Accounting and Billing Service

6.1 Accounting versus Banking

Given that accountability can be defined through terms of an SLA, it is possible to consider providing services to verify and record the consequences of accountable actions by NextGRID services and their consumers.

The simplest approach to accounting is that each time a consumer performs an operation for which somebody is accountable, the accountable person's bank account or credit card is charged the appropriate amount. However, the per transaction cost imposed by the banks, and the over-head of communication with them, are both likely to be high. This approach is therefore not usable for a large number of low-cost transactions, which is likely to be the case for many (probably most) Grid applications.

It is clear that there is no point creating an alternative banking system on the Grid. Transaction clearing in the banking system is very expensive, in part because the messages really do represent cash flows. It is therefore necessary to build in many safeguards, to ensure that the system is reliable, that the two ends of a transaction are matched, that the process is completed within a short time, that the system can roll back with no loss of data in the event of error, including hardware errors, etc. Any system that met the requirements of a Grid bank would be no more cost-effective than the existing banking system – so NextGRID should not reinvent the wheel.

However, it is then necessary to provide mechanisms for recording micro-charges that are individually too small to process through the banks, so that clients can be billed on a less

frequent basis for their usage of services. Even with larger payments that can be charged to a bank account directly, records must also be kept locally, both for informational purposes and to cross-check the bank's records. Finally, when a client pays a bill, the payment must also be recorded, so it can be taken into account in subsequent billing periods. The service to provide these capabilities we term an accounting service – a management information service for aggregating and monitoring micro-charges, that provides input from which the real financial transactions can be raised.

It is important to emphasise that this accounting service is not a banking service. It records money that a person is owed and owes, rather than money which they actually have. It does not guarantee (the way a bank has to) that these records will be 100% complete and accurate. A service operator therefore can't demand money from the operator of the accounting service, or treat the balance of an account as cash that can be used to pay others. The service operator can only send an invoice to the account holder, who may contest the details if there have been any operational or network errors. Only when they get the money through the real banking network should they record the payment and spend the cash.

6.2 Desired Behaviour

In GRIA, the accounting service provides these capabilities, but only for the service provider. The accountable consumer opens an account with each service provider, and can use its functionality to control which other consumers (e.g. project staff or collaborators) they are willing to be accountable for. The service provider uses the accounting service to track micro-payments associated with individual consumer actions, to limit the total amount of credit allowed on the account, and to provide monitoring statements (and ultimately to generate bills) for the accountable consumer.

One drawback with the existing GRIA model is that although micro-payment aggregation is handled at the service provider side, there is no aggregation on the consumer side. If the accountable consumer is running a large project, they may have to open accounts with multiple service providers. Although monitoring statements are available from each service provider, there is no support for aggregating micro-payments across all service providers.

It is therefore proposed that the NextGRID accounting service should allow both the consumer and the provider of services to maintain their own records. The service provider will record any micro-charges using its own accounting service, and communicate these to the client's accounting service so both sides can monitor what is going on. For example, in the simple orchestration scenario in Figure 4, the three sets of records might look like this:

Party	***A***	***B***	***C***
Items	Owed to B: $1.00	Owed by A: $1.00	
	Owed to C: $0.50		Owed by A: $0.50
	Owed to B: $0.80	Owed by A: $0.80	
	Owed to C: $0.40		Owed by A: $0.40
Total	-$2.70	$1.80	$0.90

Table 1. Accounting Statement Aggregation

Each service provider B and C can easily track what they are owed, and use this information to bill their customer (in this case A), as in the GRIA implementation. Unlike GRIA, the

consumer A can also track what they owe across all service providers, and use this information to check the validity of the bill they receive from each.

The other obvious drawback is that the accounting service only records charges actually made to the account, so there is no way to monitor consumer commitments (agreements to buy services) as well as completed expenditure (records of service charges). This is useful because it allows the consumer to check each item in their aggregated statement and match it to a commitment before approving real-world payment of any invoice.

The accounting service's interface should therefore provide access to the following functions:

1. Clients need to be able to open an account with a service provider.
2. Each service needs to be able to record micro-charges (debts) owed by the client.
3. Whoever handles real-world payments to the service provider needs to be able to record the receipt of payments.
4. Account services may need to contact other accounting services to aggregate information from multiple sources. For example, a client may run an accounting service in their home organisation which records what they think they owe across multiple service providers, and this must be cross-checked against the accounts the client holds with each service provider.
5. Both clients and service providers need to be able to generate a statement showing what they owe, or are owed.
6. Service providers may need to be able to reimburse a client (e.g., if a job is cancelled, or they agree that a charge was made erroneously).

Finally, a service provider needs a way to verify that an accountability assertion in an SLA is valid. This may include checking the credit limit of the account (which can normally be done by communicating with the service provider's own service), or checking that the account holder really does accept responsibility for the user requesting an action (which may require communication to a remote accounting service, or verification through other means such as the possession of a security token). We do not consider these (security-related) aspects here, as they will be addressed and supported by the NextGRID Dynamic Security components being designed in collaboration with WP4.

6.3 Accounting Service Implementation

The figure below shows the desired behaviour of accounting services in a simple scenario with one client and one service provider. The client is using a commitment accounting system, in which (an authorised) client tells its accounting service how much it expects to pay to whom for a service, and the accounting service records this commitment. When the statement is available, any charges will be reconciled against the commitment, making it easy to check the statement.

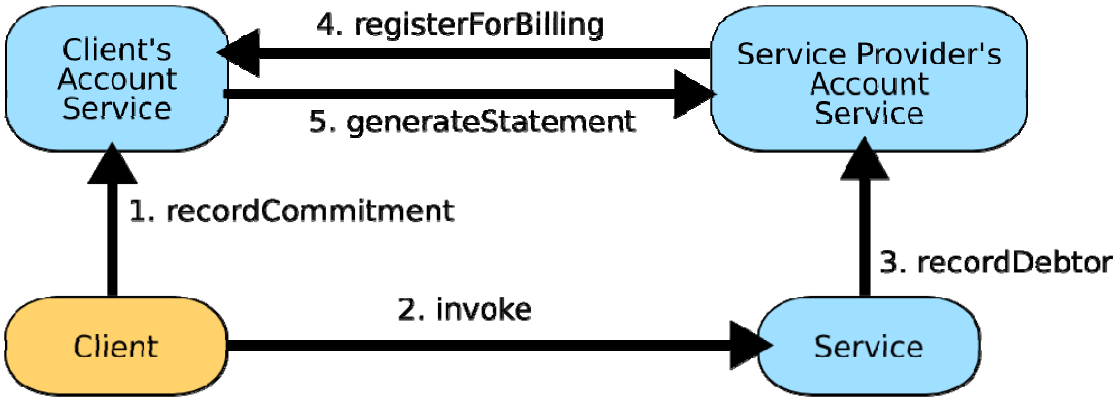

Figure 6. NextGRID Accounting Service Example

The sequence of operations show in the figure is:

1. recordCommitment(service, amount, message) -> billingToken. This opens up an authorisation for the service provider to bill to the client's account, which the client will be liable for. The billing token may contain the authorised service, the commitment ID and the amount for which the service is authorised to bill.
2. The client invokes an operation on the service, including billingToken in the header.
3. recordDebtor(client, billingToken, message). This tries to record the debt with the service provider's accounting service. The service provider's accounting service checks that the client is authorised by validating the billing token.
4. registerForBilling(account). This notifies the client's accounting service that the service provider's accounting service is recording the client's liability for use of the service. The client's accounting service then adds the service provider's account to the list of accounts on which it should periodically call generateStatement().
5. At regular intervals, the client's accounting service calls generateStatement on all of the service provider accounts for which the client is liable. It reconciles the items on these statements with the commitments it recorded when it originally issued the billing tokens.

Proposed list of key operations on accounts at the service provider:

- recordCommitment(service, amount, message). The message records the reason for the commitment, and returns a billing token (or context) to the client.
- recordDebtor(amount, billingToken, message). Called by the service each time the client incurs a new liability. The billing token here appears on the statement, allowing the client to reconcile the item. The message provides information about the reason for the liability, such as a textual message, the user who invoked the service, and the resource on the service being invoked.
- registerForBilling(account). Notifies the client's accounting service that an account is being held for them at the service provider.
- recordPayment(amount, message). Called by the service provider to notify its own accounting service of a payment received. The message indicates any information about the payment.
- getAccountStatement(period). Returns a statement showing how much is owed, and for what.

It may also be necessary for the service provider to record expected/estimated debts for long running on-going operations, and provide a way to turn this into a regular debt record once the

operation has completed. It should also be possible to support a model in which a client's payment indicates which items it is paying for, and the service provider's accounting service marks those items as paid. These are refinements which affect how information is provided to users of the accounting services, and not the fundamental interactions between them.

6.4 Remarks

These developments should be tested through implementation and execution in a range of environment that simulate the range of VO types that have been discussed here. The environment need not be a full NextGRID environment, but can be one that extracts the salient features. The aim of these experiments is to refine the architectural concepts. One possible test bed is Unicore combined with the Accounting Service specified above. Unicore can be enhanced to carry "statement of accountability" like statements (by a suitable extension of the User field), these can be interpreted as accounting operations.

7 Conclusions and Recommendations

The conclusions of this work are as follows.

1. We propose that there are 3 federation primitives that should be supported by the NextGRID architecture.
2. Delegation is an overloaded term, which has caused some problems; this can be corrected by a precise use of the term.
3. We have analysed three existing Grids (Globus, Unicore and GRIA), and conclude that the three federation primitives are more than sufficient to implement their federation models. Indeed Globus and Unicore exhibit only (dynamic) orchestration, while GRIA also has consumer federation but only static orchestration. None of the systems explicitly recognise encapsulation.
4. We have also examined how the federation models from these three current-generation Grids could be expressed through SLAs as defined by NextGRID, and found that it is feasible to do this, and to automate orchestration actions governed by them.
5. We note that all these SLAs should have clauses to prevent untrustworthy actions of delegates in service orchestration, which in these current Grids could not easily be detected by the delegator.
6. We have proposed a set of requirements for a "statement of accountability" that should be a part of every NextGRID SLA.
7. We have analysed and documented the functionality requirements for a generic accounting service.

We recommend that WP3 take account of the different types of SLA clauses we found were needed to represent behaviour and accountability in typical current Grid systems, and also the need to have SLAs that can refer to other SLAs in the same or related contexts.

We recommend that WP4 and WP5 should investigate how a more flexible dynamic security implementation could be used to reduce business dependency and thus simplify the number of non-automated "business trust" terms needed in SLAs.

We recommend that WP5 should develop a generic accounting service, based on the NextGRID dynamic security core, to form part of the common infrastructure and provide a standard way to manage accountability and create billing records in NextGRID.

The accounting service should be used in a series of experiments that examine the ways that statements of accountability can be formed and the terms that they should carry.

8 References

The following references provide further information as indicated in this report.

1. EC IST Project 511563 *The Next Generation Grid, Annex I: Description of Work, V1.0*, as authorised by the European Commission on 17 June 2004.
2. HP, IT Innovation, EMIC, Intel, *NextGRID Project Output P3.2.1: Business Models Report*, V1.0, 01 April 2005.
3. EC Project IST-2001-33240 Grid Resources for Industrial Applications. See http://www.gria.org for more details.
4. EC Project IST-2001-37153 Grid Enabled Medical Simulation Services. See http://www.gemss.de for more details.
5. *Web Service Level Agreement (WSLA) Language Specification*, published by IBM, see http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf.
6. FLE, BT, NEC (editors), *NextGRID Vision and Architecture White Paper*, V1.0, October 2005.
7. Source: *Oxford English Dictionary, 2nd Edition*, Oxford University Press, 1989.
8. V. Welch, I. Foster, C. Kesselman, O. Mulmo, L. Pearlman, S. Tuecke, J. Gawor, S. Meder, F. Siebenlist. *X.509 Proxy Certificates for Dynamic Delegation*. 3rd Annual PKI R&D Workshop, 2004. See also IETF RFC3820, http://www.ietf.org/rfc/rfc3820.txt.
9. Foster, I. (2005) *A Globus Toolkit Primer*. (Available from http://www.globus.org/toolkit/docs/4.0/key/GT4_Primer_0.6.pdf). For further details see http://www.globus.org.
10. Antonioletti, M. *et al.* (2005) *The design and implementation of Grid database services in OGSA-DAI*. Concurrency and Computation: Practice and Experience. v17, Issue 2-4 , Pages 357 – 37. For further details see http://www.ogsadai.org.uk/.
11. L. Pearlman, V. Welch, I. Foster, C. Kesselman, S. Tuecke. *A Community Authorization Service for Group Collaboration*. Proceedings of the IEEE 3rd International Workshop on Policies for Distributed Systems and Networks, 2002.
12. Chaitanya Baru, Reagan Moore, Arcot Rajasekar and Michael Wan. *The SDSC storage resource broker*.In Procs of CASCON Conference, 1998. See http://www.sdsc.edu/srb for more details.
13. Platform Community Scheduler Framework, see http://sourceforge.net/projects/gcsf/ for details.
14. Erwin, D. (Ed) 2003 *UNICORE Plus Final Report - Uniform Interface to Computing Resources*. ISBN 3-00-011592-7 (Available from www.unicore.org/documents/UNICOREPlus-Final-Report.pdf)
15. Snelling, D., van den Berghe, S. and Li, V. (2004) *Explicit Trust Delegation: Security for Dynamic Grids.* Fujitsu Sci. Tech. J., v40, 2, p282-294. (Available from http://www.fujitsu.com/global/news/publications/periodicals/fstj/.)
16. Surridge, M., Taylor, S. J. and Marvin, D. J. (2004) *Grid Resources for Industrial Applications*. In Proceedings of 2004 IEEE International Conference on Web Services, pages pp. 402-409, San Diego, USA. See http://www.gria.org for the latest version.
17. See http://java.sun.com/blueprints/patterns/SessionFacade.html. This is just one of many design patterns adopted by the J2EE community, though not yet by the Grid community.