EGEE

EGEE Middleware Architecture

AND PLANNING (RELEASE 1)

EU Deliverable DJRA1.1

Document identifier:	**EGEE-DJRA1.1-476451-v1.0**
Date:	**August 26, 2004**
Activity:	**JRA1: Middleware Engineering and Integration**
Lead Partner:	**CERN**
Document status:	**FINAL**
Document link:	**https://edms.cern.ch/document/476451/**

Abstract: This document describes the Service-Oriented Architecture of the EGEE project's *gLite* middleware. It describes the individual Grid services that constitute the EGEE middleware architecture, presents requirements on these services, and explains by means of Use Cases how these services are expected to work together to achieve the goals of the end-user. The EGEE release plan (http://edms.cern.ch/document/468699/1.0), which presents the detailed plans of how these services will be implemented in the course of the EGEE project, augments this document.

Delivery Slip

	Name	Partner	Date	Signature
From	EGEE Design Team	CERN, CCLRC/RAL, INFN, DATAMAT, CESNET, NIKHEF, KTH/PDC	01/07/2004	
Reviewed by	Charles Loomis Joni Hahkala Javier Orellana	CNRS UH.HIP UCL	22/07/2004	
Approved by	PMB		18/08/2004	

Document Change Log

Issue	Date	Comment	Author

Document Change Record

Issue	Item	Reason for Change

CONTENTS

1 INTRODUCTION

1.1 PURPOSE OF THE DOCUMENT

This document describes the middleware architecture of the EGEE middleware, called *gLite*, by means of a service oriented architecture. It is a living document and we expect modifications of the proposed architecture as we gain experience with practical implementations, feedback from our users, and evolution of the requirements.

In the remainder of this paper we discuss some key requirements for our Grid architecture in Section 3 followed by a brief introduction to the principles of service oriented architectures in Section 4. Sections 5-9 present the gLite Grid services in detail. The interactions amongst those services are explained in Section 10. Open issues are covered in Section 11. The paper concludes with a brief summary after having discussed a few implementation considerations in Section 12.

1.2 APPLICATION AREA

This document applies to the implementation of the gLite middleware within the scope of the EGEE project and the JRA1 and JRA3 activity mandate. It is also applicable to other activities within EGEE, in particular SA1, JRA4, NA3, and NA4.

1.3 DOCUMENT AMENDMENT PROCEDURE

This document can be amended by the EGEE JRA1 design team. The document shall be maintained using the tools provided by the CERN EDMS system.

1.4 TERMINOLOGY

AA	Attribute Authority
AC	Attribute Certificate
ACL	Access Control List
AFS	Andrew File System
API	Application Programming Interface
AuthN	Authentication
AuthZ	Authorization
CA	Certification Authority
CAS	Community Authorization Service
CE	Computing Element
CEA	Computing Element Acceptance
DGAS	DataGrid Accounting System
DNS	Domain Name System
DRMAA	Distributed Resource Management Architecture API
EDG	European DataGrid
EGEE	Enabling Grids for E-Science in Europe
FC	File Catalog
FPS	File Placement Service

FS	File System
FTP	File Transfer Protocol
FTS	File Transfer Service
GAS	Grid Access Service
GGF	Global Grid Forum
GRAM	Grid Resource Access Manager
GSI	Grid Security Infrastructure
GUID	Global Unique Identifier
GSM	Grid Storage Management
GUI	Graphical User Interface
HEP	High Energy Physics
HTTP	Hypertext Transfer Protocol
ISM	Information Super Market
I/O	Input / Output
JC	Job Controller
JDL	Job Description Language
JP	Job Provenance Service
L&B	Logging and Bookkeeping Service
LCG	LHC Computing Grid
LFN	Logical File Name
LHC	Large Hadron Collider
LRMS	Local Resource Management System
NAT	Network Address Translation
NE	Network Element
NTFS	(Microsoft) NT File System
OCSP	Online Certificate Status Protocol
OGSA	Open Grid Services Architecture
PAT	Policy Administraton Tool
PCI	Policy Communication Interface
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PKI	Public Key Infrastructure
PM	Package Management
POSIX	Portable Operating System Interface (X)
PR	Policy Repository
QoS	Quality of Service
RADIUS	Remote Authentication Dial In User Service
RC	Replica Catalog
RDMS	Relational DataBase Management System
RLS	Replica Location Service
SAML	Security Assertion Markup Language
SE	Storage Element
SIPS	Site Integrated Proxy Service
SLA	Service Level Agreement
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SRM	Storage Resource Manager

SURL	Site URL
TQ	Task Queue
UC	User Context
URL	Uniform Resource Locator
UUID	Universal Unique Identifier
VDT	Virtual Data Toolkit
VO	Virtual Organisation
VOMS	Virtual Organisation Membership Service
WM	Workload Manager
WMS	Workload Management System
WS	Web Services
WS-A	Web Services Addressing
WS-E	Web Services Eventing
WS-I	Web Services Interoperability
WS-N	Web Services Notification
WSRF	Web Services Resource Framework
XACML	eXtensible Access Control Markup Language

2 EXECUTIVE SUMMARY

Grid systems and applications aim to integrate, virtualise, and manage resources and services within distributed, heterogeneous, dynamic *Virtual Organisations* across traditional administrative and organisational domains (*real organisations*) [50].

A Virtual Organisation (VO) comprises a set of individuals and/or institutions having direct access to computers, software, data, and other resources for collaborative problem-solving or other purposes. Virtual Organisations are a concept that supplies a context for operation of the Grid that can be used to associate users, their requests, and a set of resources. The sharing of resources in a VO is necessarily highly controlled, with resource providers and consumers defining clearly and carefully just what is shared, who is allowed to share, and the conditions under which sharing occurs [49].

This resource sharing is facilitated and controlled by a set of services that allow resources to be discovered, accessed, allocated, monitored and accounted for, regardless of their physical location. Since these services provide a layer between physical resources and applications, they are often referred to as *Grid Middleware*.

The Grid system needs to integrate Grid services and resources even when provided by different vendors and/or operated by different organisations. The key to achieve this goal is standardisation. This is currently being pursued in the framework of the Global Grid Forum (GGF) and other standards bodies.

In this document we present the architecture of the EGEE Grid Middleware (called `gLite`). It is influenced by the requirements of Grid applications (cf. Section 3), the ongoing work in the Global Grid Forum (GGF) on the Open Grid Services Architecture (OGSA) [50], as well as previous experience from other Grid projects such as the EU DataGrid (EDG) (`http://www.edg.org`), the LHC Computing Grid (LCG) (`http://cern.ch/lcg`), AliEn (`http://alien.cern.ch`), the Virtual Data Toolkit VDT (`http://www.cs.wisc.edu/vdt/`) (including among others Globus (`http://www.globus.org`) and Condor (`http://www.cs.wisc.edu/condor`)) and NorduGrid (`http://www.nordugrid.org`).

The `gLite` Grid services (cf. Sections 5-9) follow a *Service Oriented Architecture* (cf. Section 4) which will facilitate interoperability among Grid services and allow easier compliance with upcoming standards, such as OGSA, that are also based on these principles. The architecture constituted by this set of services is not bound to specific implementations of the services and although the services are expected to work together in a concerted way in order to achieve the goals of the end-user (see Section 10 for some simple use cases) they can be deployed and used independently, allowing their exploitation in different contexts.

The gLite service decomposition has been largely influenced by the work performed in the LCG project (the requirements and technical assessment group on an "architectural roadmap for distributed analysis" (ARDA) [7]). Figure 1 depicts the high level services, which can thematically be grouped into 5 service groups.[1]

These services (which might internally be split in further services as described in Sections 5-9) are characterised by the scopes and enforcement of their policies as shown in Table 1. We distinguish between *user*, *site*, *VO*, and *global* (i.e. multi-VO) scope where combinations are possible (authorization policies may for instance be enforced by the VO and the site).

Although most services are managed by a VO, there is no requirement of having independent service instances per VO; for performance and scalability reasons service instances will in most cases serve multiple VOs.

Security services encompass the Authentication, Authorization, and Auditing services which enable the identification of entities (users, systems, and services), allow or deny access to services and resources,

[1] Other categorisations (e.g. a layered architecture as discussed in [27]) are possible as well.

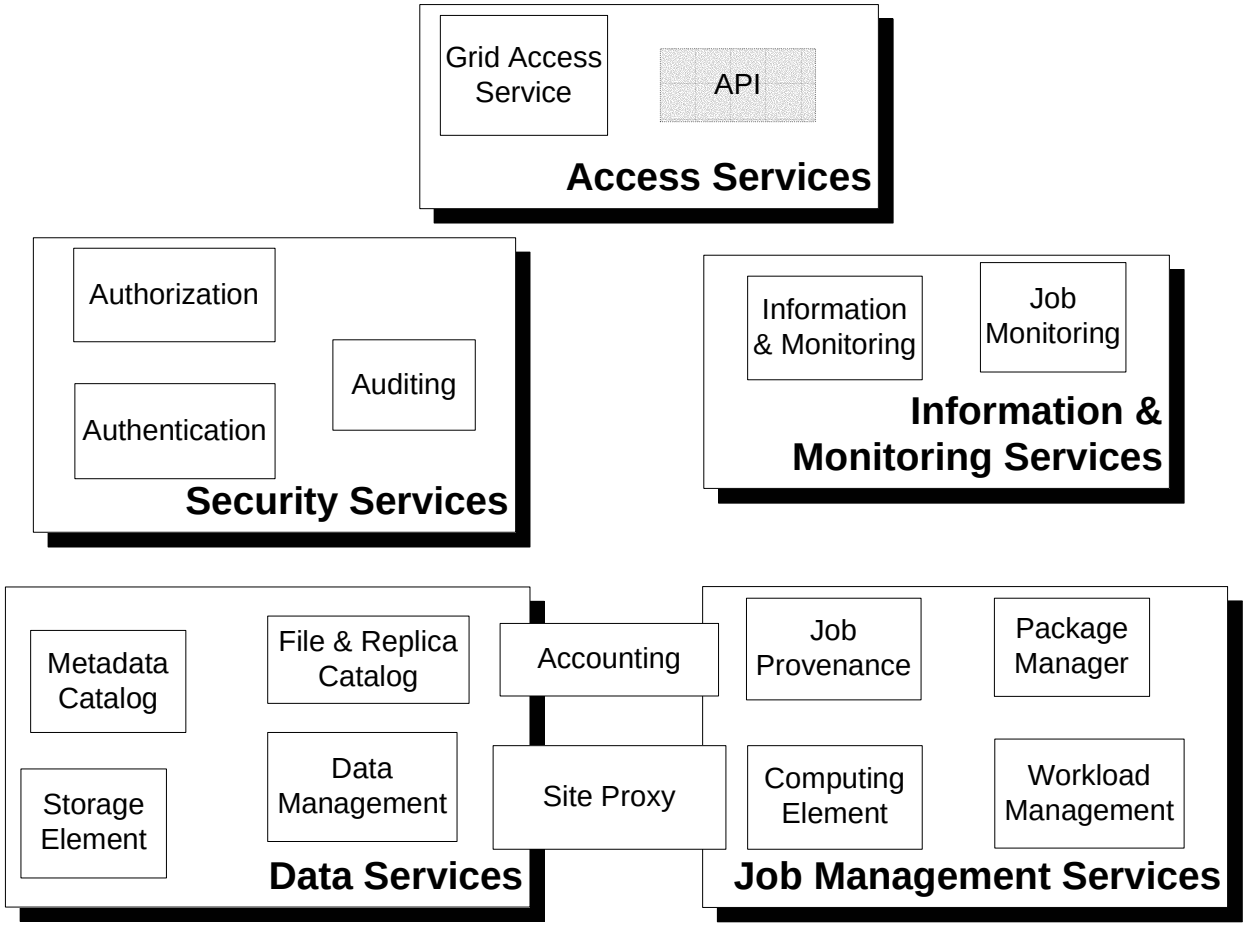

Figure 1: gLite Services

and provide information for post-mortem analysis of security related events. It also provides functionality for data confidentiality and a Site Proxy, i.e. a means for a site to control network access patterns of applications and Grid services utilising its resources.

API and Grid Access Service provides a common framework by which the user may gain access to the Grid services. The access service will manage the life-cycle of the Grid services available to a user, according to his/her privileges. The API is of course not a service on it's own, however we depict it in Figure 1 to make this way of accessing the services explicit.

Information and Monitoring Services provide a mechanism to publish and consume information and to use it for monitoring purposes. The information and monitoring system can be used directly to publish, for example, information concerning the resources on the Grid.

More specialised services, such as the Job Monitoring Service, will be built on top. The underlying information service will be able to cope with streams of data and the merging and republishing of those streams. The system relies upon registering the location of publishers of information and what subset of the total information they are publishing. This allows consumers to issue queries to the information system while not having to know where the information was published.

However all published information carries with it the time and date when it was first published (i.e. when

Service	Scope
Accounting	global, VO, site
Auditing	VO, site
Authentication	global
Authorization	VO, site
Catalogs (file, replica, metadata)	VO
Computing Element	VO, site
Data Management	VO, site
Grid Access Service	VO
Information & Monitoring	global, VO, site
Job Monitoring	user, VO
Job Provenance	user, VO
Package Manager	VO
Storage Element	VO, site
Workload Management	VO
Site Proxy	global, VO, site

Table 1: gLite Services and their Scope

the "measurement" was made) as well as the identity of the publisher and from where it was published. This information is not modifiable even if data are republished. In fact no data can be modified in the system thus avoiding any inconsistencies when data are republished. There are of course mechanisms to clean out old data (under the control of the publisher of that data).

Finally there is a fine-grained, rule-based, authorization scheme to ensure that people can only read or write within their authority.

Job Management Services The main services related to job management/execution are the computing element, the workload management, accounting, job provenance, and package manager services. Although primarily related to the job management services, accounting is a special case as it will eventually take into account not only computing, but also storage and network resources. Hence, accounting is depicted in Figure 1 straddling the data and job management services.

These services communicate with each other as the job request progresses through the system, so that a consistent view of the status of the job is maintained.

When this communication needs to occur during execution on the computing nodes (notably in the cases of communication to the job provenance service, to the package manager service, and to any network service that provides "interactive" services such as the bridging and buffering of standard streams, or signalling to running jobs), the usual technique of wrapping the executable content into "wrapper" scripts will be used. This technique, while not constituting a "service" as defined in Section 4, can be used at various levels in the architecture, as different parties, such as the submitting user and the workload management service itself, require the output of procedures that are inserted in the wrapper. When job wrapping requires services to be created or accessed by the submit/user interface node (before the service port for job submission is contacted), tools will be provided to make this operation transparent to the user.

Data Services The three main service groups that relate to data and file access are: Storage Element, Catalog Services and Data Management. Closely related to the data services are the security-related services and the Package Manager.

In all of the data management services described below the granularity of the data is on the file level. However, the services are generic enough to be extended to other levels of granularity. Data sets or collections are a very common extension where the information about which files belong to a dataset may be kept in an application metadata catalog. The usual possibility to group files by virtue of directories is provided. Most application data is expected to be located in files (as opposed to relational database systems for example). For application metadata we don't make this assumption. In a distributed environment, there will be many replicas (managed copies) of the user's files stored at different physical locations. To the user this may be totally transparent, the middleware will provide the capabilities for replica management. The Data Scheduling services will expose all nontrivial interfaces to the user for data placement in a distributed environment.

We expect the applications to specify the files they intend to access during their jobs either by name or by metadata query. The Metadata Catalog is application (and VO) specific, so the interface we specify in this document is a very simple one that can be implemented on top of existing application metadata services such that they can be used from within the EGEE Grid environment. It is this mechanism that enables the concept of virtual datasets as well.

To the user of the EGEE data services the abstraction that is being presented is that of a global file system. A client user application may look like a Unix shell (as in AliEn) which can seamlessly navigate this virtual file system, listing files, changing directories, etc.

The data in the files can be accessed through the Storage Element (SE). The access to the files is controlled by Access Control Lists (ACL).

The detailed semantics of file access will be different depending upon what kind of storage back-end is being used beneath an SE; there may be substantial latencies for reads and a large number of possible failure modes for write.

The Storage Element, File, Replica and Metadata Catalog and Data Management services are discussed in Section 9.

Note, that the gLite architecture does not in general impose specific deployment scenarios (i.e. how many instances of a certain service are available to a user, if a service is replicated or distributed, etc.) unless specifically pointed out in Sections 5-9. Most importantly, service instances may serve multiple VOs which will facilitate the scalability and performance of the Grid system although a VO may require its own instance as well.

3 REQUIREMENTS

Due to the heterogeneous nature of Grid environments and Grid applications, a Grid architecture needs to cope with a large set of (sometimes conflicting) requirements. The OGSA document [50] gives a good overview on the nature of these requirements which are summarised below.

Heterogeneous Environment Support In general, Grid environments tend to be heterogeneous and distributed, encompassing a variety of operating systems, hosting environments, and devices. Moreover, many functions required in distributed environments may already be implemented by stable and reliable existing legacy applications which thus need to be integrated into the Grid. In order to support these requirements, resource discovery, resource interrogation, and life-cycle management of hosting environments are essential tools.

Resource Sharing Across Organisations One of the main purposes of Grid technology is to utilise resources transparently across administrative domains. Virtual Organisations provide the context to associate users, requests, resources, policies, and agreements without being limited by barriers such as existing sites or organisations. The Grid architecture needs to provide appropriate VO management, VO security, and accounting tools.

Resource Utilisation Grid environments must cater to chaotic usage patterns, making an optimal planning for resource utilisation virtually impossible. Nonetheless statistical and local optimisation of resource utilisation are still important; therefore tools for reservation, metering, monitoring, and logging are required so that system administrators can manage resources effectively.

Job Execution The Grid environment must enable submitted jobs to have coordinated access to VO resources and provide functions (such as job monitoring, analysis and projection of resource usage, dynamic adjustments) so that jobs can receive their desired quality of service (QoS). This requires tools for scheduling, job life-cycle management, work-flow management, and service-level agreement (SLA) management.

Data Services An ever-larger number of fields in science and technology require efficient processing of huge quantities of data. In addition, data sharing is important, for example to enable sharing information stored in databases which are managed and administered independently. A Grid architecture needs to provide services for data access, integration, provisioning, and cataloging.

Security Safe administration requires controlling access to services and resources through robust security protocols and according to security policies. Thus, mechanisms for authentication, authorization, and auditing are required. Moreover, the Grid architecture should work across firewalls as well as allow for network isolation, delegation, and cross-organisational policy exchange and management.

Administrative Cost The complexity of administration of large-scale, distributed, heterogeneous systems increases administration costs and risks of human errors. Support for administration tasks should thus be automated, in particular for provisioning, deployment and configuration, application management, and problem determination.

Scalability Many Grid applications are concerned with scalability and performance, in particular through distributed supercomputing and high-throughput computing applications.

Availability Virtual Organisations enable transparent sharing of alternative resources across organisations as well as within organisations, and thus can be used as one building block to realise stable, highly-reliable execution environments. This requires mechanisms for disaster recovery and fault management.

Application Specific Requirements Apart from these generic requirements, Grid applications may have specific requirements that a Grid architecture needs to fulfil. In the context of EGEE the needs of two pilot user communities must be satisfied: High Energy Physics (HEP) and Biomedical communities.

The detailed requirements, which are mostly specialisations of the generic requirements presented above, can be found in several documents:

- Common use cases for HEP [9]
- Common Use Cases for a HEP Common Application Layer for Analysis [10]
- Joint list of use cases from HEP, Biomedical, and Earth Observation [21]
- Biomedical requirements [22].

4 SERVICE ORIENTED ARCHITECTURE

Traditionally, applications have been built for a single computational entity, by integrating local system services like file systems and device drivers. Since everything is under local control, this model is very flexible in providing access to a rich set of development resources and provides precise control over how the application behaves. At the same time, this is bound to a single operating system and architecture, which can be error prone and costly, especially for upgrades. For distributed communities, it is not always possible to work in this mode with a single supercomputer as the computing entity.

A more modern approach is to construct complex distributed applications by integrating existing applications and services across the network, adding data entities, facades[2] and business logic. The aim is to reduce development time and increase productivity and software quality. This architectural model is powerful in the sense that it is very flexible and extensible, providing added functionality to the users. However, subsequent modification and architectural reuse of the components may be problematic as it may have complex repercussions on services built on top of them.

The term Service Oriented Architecture (SOA) is increasingly used to refer to a discipline for building reliable distributed systems that deliver application functionality as services with the additional emphasis on loose coupling between interacting services [45, 8]. Tight coupling makes it hard for applications to adapt to changing requirements, as each modification to one application may force developers to make changes in other connected applications.

A SOA significantly increases the abstraction level for code re-use, allowing applications to bind to services that evolve and improve over time without requiring modification to the applications that consume them. Services provide a clean model to integrate software systems both inside the organisation and across organisational boundaries. This model is clearly very suitable for Grid middleware, which has to deal with Virtual Organisations and Site policies.

The services communicate with each other through well-defined interfaces and protocols, which can involve simple data passing or some coordination of activities.

[2] Provide a unified interface to a set of interfaces in a subsystem. A facade defines a higher-level interface that makes the subsystem easier to use. [20]

4.1 SERVICES

If a service-oriented architecture is to be effective, we need a clear understanding of the term service. A service is a function that is well-defined, self-contained, and does not depend on the context or state of other services. A Web service is an application that exposes its features programmatically using standard Internet protocols.

In more detail, services are discrete units or modules of application logic that expose message-based interfaces suitable for access across a network. Typically, services provide both the semantics and the state management relevant to the problem they address. When designing services, the goal is to effectively abstract and encapsulate the logic and data associated with real-world processes, making intelligent choices about what to include and what to implement as separate services. Well-written services expose a semantically simple logic to the application that binds to them, with clean failure modes.

In a distributed environment, services tend to be oriented toward use over a network by other services, though this is not an absolute requirement. Communicating in a distributed system is intrinsically slower and less reliable than when operating in the same processing environment. This has important architectural implications because distributed systems require that developers (of infrastructure and applications) consider the unpredictable latency of remote access, concurrency issues, possibility of partial failure, and inaccessibility of certain services.

4.2 MESSAGES

Services interact by exchanging messages. Ultimately, every service is defined purely by the messages it will accept and produce, and what happens on failures. The routing of messages between services is a complex process that is best handled by a common messaging infrastructure.

Messages are sent in a platform-neutral, standardised format defined by the service interfaces. Service-to-service communication follows the interface contract; by making this contract explicit it is possible to change one service implementation without compromising the interaction. The internal structure of a service, including features such as its implementation language are, by design, abstracted away in the SOA.

For Web services, the Simple Object Access Protocol (SOAP) standard [35] is used to specify how the messages are being passed between services, and the Web Service Definition Language (WSDL) is used to specify the interface a service exposes. From a WSDL document a client or application will know how to bind to a service. Many semantic details are only defined as part of the internal logic and cannot be programmatically exposed; these details must be throughly documented.

4.3 POLICIES

All services are also governed by policies. Policies are less static than business rules and may be regional, organisational or user-specific. Policies are usually applied at run-time.

Policies may represent security, quality-of-service, management, and application concerns. Services negotiate using policies. Services must comply with one another's policy requirements in order to interoperate.

In a Grid the responsibilities for the management of the service policies is shared between many organisations and sites. Policies are typically managed by the Virtual Organisations (VO) using the service and the site administrators of the site where the service is actually deployed.

The breadth of the policy management rules needs to be considered at the design phase of the project as later changes to the system are difficult. If the policy-enforcement infrastructure is well-integrated, it al-

lows setting, changing, and evolving the run-time rules that govern communication and service behaviour easily.

4.4 STATE

Services manage state, ensuring through their logic that the state is kept consistent and accurate. State manipulation is governed by the internal semantics of the service, also called business rules. These are relatively stable algorithms and are typically implemented as part of the service application logic.

Almost all services manage durable state; that is, state that is stored on some durable medium such as a file system or in a database. The services receive a request from another service, retrieve some state from that durable medium, and build a response or update the state. This durable state is important; services may be brought down and when they are brought up again, the durable state is still there and they can continue as if nothing has happened. Services do their best to keep that durable state consistent; they would like to keep their application state in memory consistent as well, but if something happens, they can just abort the processing, forget their memory state, and set up again using the durable state.

So in summary, a complete definition of services might be: Services are network-capable units of software that implement logic, manage state, communicate via messages, and are governed by policy.

5 SECURITY SERVICES

Security services encompass the Authentication, Authorization, and Auditing services which enable the identification of entities (users, systems, and services), allow or deny access to services and resources, and provide information for post-mortem analysis of security related events.

5.1 AUTHENTICATION

Authentication is concerned with identifying entities (users, systems, and services) when establishing a context for message exchange between actors. Such information is used in many policies for resource access and data protection, as well as for auditing and incident response.

One of the key aims for Grid authentication is enabling single sign-on for the user – and a single identity credential with “universal” value – across many different infrastructures, different communities and virtual organisations, and multiple projects. Here, particular attention must be paid to the fact that the same identity is also to be used for accessing non-grid resources such as networks[3] and web resources.

This architecture allows identities to be issued by grid identity providers, by independent non-grid entities (like governments or other national agencies), and by users’ home organisations (major institutions and laboratories). Thus, authentication (identity) assertions are independent of any authorization assertions and policies – there is no implied right-of-access associated with an identity.

Although the most used model for Grid security to date is based on the end-entity (user) holding the authentication credentials, this is by no means the only option available. In many systems, the user may attempt to access a service directly, and if authentication is required a security exchange will be initiated. Moreover, such a model allows for implicit privacy preservation: if no authentication is required that step may be omitted and the security context established based on authorization data only.

Alternatively the end-user may have a specific identity and the capability to obtain credentials, but they are either kept on behalf on the user by some other entity (typically the user’s home organisation using the NERSC secure MyProxy) or implicitly in the user’s infrastructure (site-integrated credential services such as the Kerberos Certification Authority).

[3] see Section 11.5 for a discussion on networks and the Grid

The VO is not the appropriate guardian for the user's credentials, because this implies a proliferation of those credentials to many different entities (with the associated risks of compromising those credentials). Moreover, it entails the risk that this binding to the VO inadvertently leads to identities and authentication tokens being linked to a single VO, thus maxing authentication and authorization.

Nevertheless, the model where the user, at least theoretically, owns the credentials has most leverage in current grid-security implementations. The model allows for straightforward credential mobility, external (governmental) identity providers, and single sign-on capability over multiple domains even in the absence of any direct trust relationship between those domains.

This system is based around a Public Key Infrastructure (PKI) built around trusted third parties (Certification Authorities or CAs). Single sign-on capabilities can be obtained by generating a proxy of the user's credential that is not protected by further activation data. That proxy credential may be based on a long-term credential held by the entity that should be protected by activation data. The proxy should be able to contain any assertions that are needed for establishing the security context when required by the session establishment protocol.

Thus a PKI based authentication system with proxy credentials (GSI[25]) will be the starting point for developing the authentication services. The user will be presenting his or her own identity credential proxies to the services, including any attribute assertions and relevant policies, and whenever actions are taken on the user's behalf these credentials will be transferred to the service invoked in order to retain secure traceability. Services that act, directly or indirectly, on behalf of, or as a result of, action by the user, shall hold the credentials derived from the credentials of the original user.

5.1.1 TRUST DOMAINS

The use of a PKI implies the existence of third parties (Authorities) that are considered "trusted" by all participants of the Infrastructure. Traditionally implemented by a hierarchical structure, the absence of a single global root of trust for X.500 compels the use of a different structure. For the European (and related world-wide) scientific Grid communities, a single trust domain has been established based on a policy defining a common set of minimum requirements. A policy management authority (the EUGridPMA[15]) has been established to coordinate this trust domain.

The trust anchors pertaining to the domain must be accessible to the services that validate authentication data in a secure, non-tamperable location (e.g. a local disk, modifiable only by the administrator).

5.1.2 SITE-INTEGRATED PROXY SERVICES

Site-integrated proxy services (SIPS) are services that generate user proxy credentials without the user holding any long-term credentials. Several SIPS implementations exist (such as kCA[41], and the Virtual Smart Card project [11]) and are described extensively elsewhere.

5.1.3 REVOCATION

Timely revocation of identity is needed to prevent exploitation of credentials that have been compromised. The allowed response time as specified by resource providers is in the region of 10-60 minutes. This constraint cannot be satisfied by the periodic distribution of validity/revocation lists.

Any software component that validated authentication must be able to check the validity of the credentials in real-time, using a suitably-sized mesh of status responders. The protocol for validating authentication credentials will be the Online Certificate Status Protocol, OCSP [14]. As a backup mechanism in case of network partitioning, revocation lists should be distributed periodically (4-6 times per day) and retrieved by all relying parties (users and service providers).

5.1.4 CREDENTIAL STORAGE

The usage pattern for credentials by users and services is slightly different. Services will keep credentials stored locally with the service but typically without activation data. Those services that are created on-demand (or on behalf of a user or process on a remote system) should use delegated credentials from either the originating system or the user responsible. Such a store of delegated credentials is to be managed as a collection of resources (delegation is described in the next section).

On the other hand, users are mobile and their credentials are not only more exposed but also much more "powerful" and attractive to exploit. Thus, long-term credentials held by the user must be stored securely and be protected by a passphrase. Several alternatives are to be provided: another trusted organisation may hold the credentials on the user's behalf, the credential may be stored on a smart card, or the user may not actually posses a long-term credential, but proxy credentials are created on-demand. Storing credentials on a local file system should be supported, but implies a significant security risk.

In either case, the credentials must be issued by a trusted party close to the user (a certification authority, or the user's home organisation) and valuable credentials held in a secure storage under the user's control.

The user will likely have different ways of obtaining credentials, and may have more than one credential at the same time. Thus, a "credential wallet" function should be provided (e.g. MyProxy). As mentioned earlier, this credential wallet is valuable to the user and may contain sensitive personal data. It must be located where the user can trust the store (e.g. with the user's home organisation).

At all times, the proliferation of long-term authentication credentials should be traceable, so that all copies of the credential can be disabled or revoked when the credential is compromised.

5.1.5 PRIVACY PRESERVATION

One of the basic security requirements in the Grid is traceability of actions across domains. However, linking those actions to actual identities or to long-term reusable credentials exposes a significant amount of personal data that could violate privacy requirements. Moreover, the very actions themselves may reveal commercially valuable information that should have remained hidden.

These issues may be addressed by the introduction of independent pseudonym providers. Such pseudonym providers issue identity tokens in the same format as CAs in the common trust domain, but do require that a new trust domain be established between all parties involved.

The pseudonym provider must keep a secure, non-tamperable and private account of the associations between issued pseudonyms and the original identities if the trust domain requires traceability and auditability of actions.

5.1.6 SECURITY CONSIDERATIONS

Credential stores, such as MyProxy, are high-value targets for directed attacks when used by many users. Hosting such services requires a trusted computing environment with dedicated machines, active firewalls, and minimal services. Moreover, these services should be compartmentalised so as not to store too many valuable credentials at the same time. Thus, the credential store must never be run as a common service by a VO, since a successful attack on the credential store will disable the entire VO for a long period of time. Hosting this service with the user's home organisation will both increase the trust level of the user in this service, as well as limit any damage by compromise to that single organisation.

5.2 AUTHORIZATION

Authorization (AuthZ) is concerned with allowing or denying access to services based on policies. The core problem with authorization in a Grid setting is how to handle the overlay of policies from multiple administrative domains (user policy, VO specific policy, operational procedures, site-local policy), and how to combine them.

There are three basic authorization models [43], classified as *agent*, *push* and *pull*. In the *agent* model, an authorization service issues tokens. The user collects the tokens and presents these to the resource where access is requested. While putting additional burden on the client, the resource does not need to know ahead of time about the user's privileges.

In the *push* model, the user only interacts with the AuthZ server, which in turn forwards service-specific parts of the request to the underlying resources. While being a centric approach, network bandwidth or connectivity provisioning is best done in this mode, since access to the network in the end must be transparent.

In the *pull* model, the resource asks the AuthZ service on a need-to-know basis. This puts the burden on the resource, as it needs to know up-front all authoritative parties in the system, and how to contact them. Cellular phone roaming, and various RADIUS-based [12] network access services use this model.

The *agent* model has been identified as the model that best suits dynamic, distributed and loosely coupled systems such as Grids.

There are two kinds of AuthZ services: attribute authorities (AA), which associate a user with a set of attributes in a trusted manner to a relying party, by way of digitally signed assertions. The relying party (the resource) evaluates the attribute assertions (e.g. role and group membership information) and include it as "evidence" when evaluating an access request against local policy.

The other kind of AuthZ service deals with policy assertions: here, the resources consolidate some of the the policy definition to a trusted third party. The policy assertion service will issue claims that gives a user (or set of users) the explicit privilege to perform an action (or a set of actions) on a certain resource (or set of resources). The resource owners in turn may or may not decide to comply with these claims, pending the evaluation of conflicting local policy.

5.2.1 DELEGATION

It is often the case that Grid users need to delegate some subset of their privileges to another (dynamically created) entity on relatively short notice, and only for a brief amount of time. For example, a user needing to move a dataset in order to use it in a computation may want to grant to a file transfer service the necessary rights to access the dataset and storage so that the service can perform a set of file transfers on the user's behalf. Since such actions may be difficult to predict, having to arrange delegation ahead of time is prohibitive.

An important security aspect in regards to delegation is the principle of least privilege: you only want to delegate necessary privileges, and no more. This is hard to accomplish in reality, but our architecture must support some simple provisioning and enforcement of restricted delegation. It is also often very useful to separate the privilege to delegate from the privilege itself.

A number of existing mechanisms could satisfy the delegation use case mentioned above. The use of Proxy Certificates [48] has been the mechanism most widely adopted by the Grid community, as the technology needs no additional infrastructure services, and at the same time it also solves the single sign-on and dynamic entity identification problems. Besides providing some coarse-grained controls (such as describing whether all or none of a user's privileges are implied in the delegation, and the right to delegate further), there is also a placeholder for adding arbitrary, typically application-specific, policy restrictions.

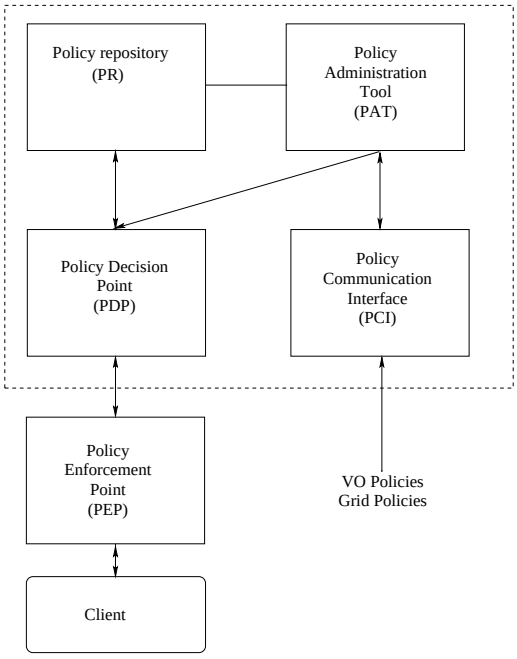

Figure 2: Architecture of the policy combination and authorization framework.

5.2.2 POLICY COMBINATION AND EVALUATION

When making an AuthZ decision, we must be able to combine information from a number of distinct sources. Besides Grid-wide or VO-wide attribute and policy assertions, potentially provided to a service as part of the client request, we also need to include domain specific policy such as POSIX ACLs for file access, or a (sub-)set of VOs that are allowed access to a particular computational resource. Finally, but most importantly, we must also take any locally defined site policy into consideration when making a unified, context specific AuthZ decision on an individual request basis.

In order to be able to combine information from multiple sources in this manner, we must have a way to convert any domain specific access control language into a common language with strong support for combining policies. One such candidate language is XACML (eXtended Access Control Markup Language) [32].

Figure 2 shows the architecture of a general framework that is able to fulfil the requirements set out above. It allows the definition and enforcement of local, VO and Grid wide policies. The *Policy Administration Tool* (PAT) allows the local administrator to specify, remove, modify policies, and accept or refuse policies proposed by an external entity (for example policies defined at the Grid level or set by the VO administrators). The *Policy Communication Interface* (PCI) is used to handle the communication between different “entities”, for example to send VO-level policies to the various CEs. The set of active (and past) policies, possibly described in the combined form using standard languages such as XACML, is held in the *Policy Repository* (PR). In this architecture the *Policy Enforcement Point* (PEP) is used by the client (e.g., the Web Service) to make the policy evaluation of a request. The PEP assembles the necessary *evidence* (asserted attributes, and other contextual data), and then forwards the request to a *Policy Decision Point* (PDP) which evaluates the request according to the active policies.

We note that for performance reasons, the PEP and PDP will in most cases be merged into a single component that is embedded in the request handling flow of the service container, rather than being exposed as separate, standalone Web Services.

5.2.3 MUTUAL AUTHORIZATION

There are Use Cases where a client wants to authorize a service as well as the service authorizing the client access: for instance, a client wanting to store sensitive data might first ensure that the SE has been approved to service such storage requests. Instead of building this tightly into every service (for instance, by having services authenticate with proxy certificates with embedded authorization information), optional means (e.g. portType) must be provided by the services to interrogate the service about such authorization information.

5.2.4 OTHER SERVICES USED

While it is possible to provide anonymous AuthZ tokens (such as tickets to a ball game), attribute assertions and policy statements are usually tied to a globally known identity for convenience, traceability and ease of management. Therefore, the AuthZ services will have a strong dependency on the Authentication infrastructure which provides the necessary unique identifiers and the cryptographic methods with which the entity-identity association can be challenged and proved.

Whenever a policy statement or an attribute assertion is created or used as evidence, a trace of this action must be found in the system. Thus, AuthZ services and policy evaluation engines will depend on and make use of available audit and logging services.

5.2.5 SERVICES

In our architecture, we foresee the need for both the AA and policy statement AuthZ services mentioned above, and the use of primarily the agent model. Initially, an AA service will provide coarse-grained division of users into different groups, which can then be handled by the resources through local configuration. However, with an increasing number of VOs using the EGEE resources, such configuration becomes unmanageable and we therefore will need a consolidation of VO policy that can be provisioned to the resources, e.g. on a need-to-know basis via the client requests. An example of a policy statement service is the Community Authorization Service (CAS) [37], which issues its statements encoded in SAML [33].

Note: we do not consider the VO policy server component to be a crucial piece of the core architecture, but we mention it anyhow at this point as it is on the future road-map.

The services and functionality to be provided are

- The VO Membership Service (VOMS) [2], is an attributes issuing service which allows high-level group and capability management and extraction of attributes based on the user's identity. VOMS attributes are typically embedded in the user's proxy certificate, enabling the client to authenticate as well as to provide VO membership and other evidence in a single operation.
- Policy combining engine, capable of gathering evidence and policy from multiple sources, and in multiple formats, and to combine these when making a final AuthZ decision.
- Mutual authorization capabilities on the service: for instance, a WSDL portType.

5.2.6 SECURITY CONCERNS

It is vital that all components of the AuthZ infrastructure are as securely managed and operated as the corresponding parts of the authentication infrastructure. For instance, the repository containing the trusted signatures of the AAs need to be as protected against infiltration as the repository of trust anchors (CAs), e.g. by distributing these trust anchors via an authenticated web site, possibly cross-checked against widely available public media.

5.3 AUDITING

Auditing is a very general term: here, we primarily mean the system security aspects of auditing, such as monitoring and providing for post-mortem analysis of security related events.

In computational Grids, auditing goes hand in hand with accounting, as they share the base requirements on the system's logging capabilities: *who did what, where, and when* (and in the case of accounting, *for how long, or for how much*). [23]

Auditing is not only meant for emergency use, such as a system breach, but is also useful for the validation of continuous operations, verifying that components behave as expected.

5.3.1 SERVICES USED

In an auditing scenario, we will make use of whatever information sources we have at hands (system logs, L&B services, ...), and attempt to make sure the other middleware components log the necessary information.

5.3.2 SERVICES

The auditing process in itself does not require a separate service. Rather, it imposes a set of common principles and practices on all system components, e.g. to log correct, complete and relevant information. The audit information should itself be traceable and verifiable throughout its lifetime and throughout service invocations, so that any anomalies in the audit trail can be traced to identifiable service components.

Creating a "real" audit service would clearly have benefits in terms of enforcing uniformity of information, providing secure remote access to the information, easing the process of merging and combining the information from different services. Although no specific architecture for such a component in currently pursued, we explicitly acknowledge the possibility to include such a service in the future, and will ascertain that it can accommodate existing or emergent services.

5.3.3 SECURITY CONSIDERATIONS

Ideally, for a system to have a "proper" auditability according to recognised standards, we would need to invest heavily in detailed documentation of processes and methodologies used, as well as employ additional technology (for example tamper-proof logs). This is more than we can consider in this project.

The audit information stored anywhere in the system should be self-contained and interpretable even if the original source of the audit information is later compromised, or if it is to be interpreted without access to the originating service.

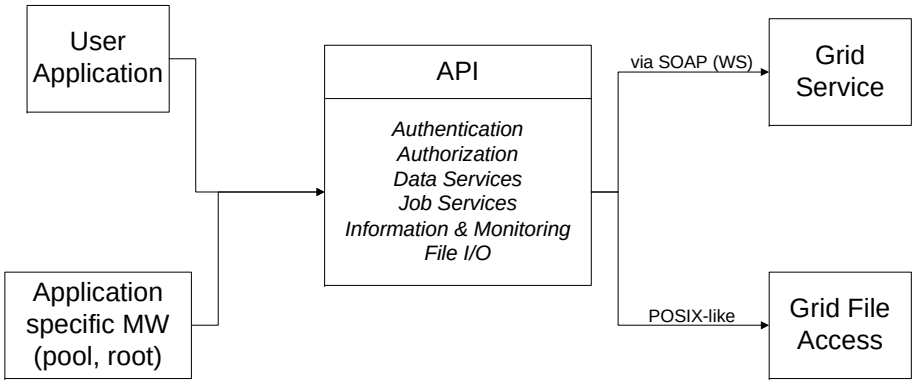

Figure 3: Grid API

6 API AND GRID ACCESS SERVICE

We envision several means by which users and application frameworks will gain access to Grid services. An API, shown in Figure 3, would be a library of functions used for building client applications, graphical user interfaces or even Grid Web portals (e.g. AliEn [6], Clarens (`http://clarens.sourceforge.net/`, or Genius [3]). The API is used also to authenticate users to the Grid, let them submit jobs, inquire job status and manage jobs, access the files available on the Grid as well as to put files onto the Grid. By files available on the Grid we understand those stored on one or more Storage Elements and registered in the File Catalog or replica location service.

The application should be able to gain access to these files by issuing requests to copy files to local temporary storage or via POSIX like interface to a nearby Storage Element.

One particularly important core service is the counterpart of the user API: the Grid Access Service (GAS). The Grid Access Service represents the user entry point to a set of core services. When a user starts a Grid session, he establishes a connection with an instance of the GAS created by the GAS factory for the purpose of this session.

Before attempting to connect to the Grid, a user is expected to register his or her temporary credentials with VO independent credential wallet (such as the NERSC secure MyProxy).

As a first step in connecting to the Grid, the user application uses the API to connect to a configurable list of Configuration Services. These are public services that can exist per VO or serve multiple VOs. They use the VO configuration, the Information Services and DNS information to deliver initial configuration to the user API. The configuration information contains the specific service endpoints which should be used by the application to connect to the Grid.

The application then connects to the GAS factory and passes over the secure line the user name and password needed to get delegation of user credentials from the credential wallet. If this operation is successful, the GAS factory will start a GAS for the user and return the service endpoint. The application then connects to its endpoint and gains access to other Grid services.

During the creation of the GAS the user is authenticated and his rights for various Grid operations are checked against the Authorization Services. The GAS keeps the user credentials and authorization information. Its lifetime will be restricted to the lifetime of delegated proxy credentials and will be managed by the user who will be able to destroy his own GAS instance. Many of the User Interface API functions are simply delegated to the methods of the GAS. In turn many of the GAS functions are delegated to the appropriate service.

This service will be constructed out of service components that will in turn present a uniform public interface to underlying services. The service components are realised as a pluggable library with each

component providing an interface to the specific middleware service. The intention is to define a sufficiently rich set of interfaces exposed to the end user to allow them to interface their application to the Grid while at the same time keeping this interface reasonably stable and protected from changes in the middleware. The components from which the interface is constructed could be to some extent defined by the VO preferences at run time (File or Metadata Catalog for example might be different for different VOs).

The GAS model of accessing the Grid is in many ways (authentication, proxy service) similar to various Grid Portals but it is meant to be distributed (the GAS factory can start GAS in a service environment close to the user in network terms) and is therefore more scalable and resilient. As opposed to a traditional Web Portal, the GAS interface is more dynamic and reflects the role of the user in the system. The GAS offers no presentation layer as it is intended to be used by the application and not by the interactive user. The traditional Grid Portal can be easily constructed by specialising GAS into a Portal Service that will provide necessary content to a presentation layer provided by the Web Server. Similarly, the specialised application services can be constructed by extending GAS or providing appropriate Service Components.

Note that all Grid services also have to be accessible directly using their respective mechanisms (i.e. not via the GAS).

Other Services Used Information Service, Authorization Services.

7 Information and Monitoring Services

7.1 Basic Information and Monitoring Services

The information services are a vital low-level component of any grid; most services will publish or consume information. The information service should create the appearance of a single federated database covering all available information. This implies a query language that allows arbitrary correlations between information in different services. A candidate for the model is the relational model with SQL as the query language. This is not to suggest that end users need to construct complex SQL queries themselves as it is more common to have front ends to automate the generation of SQL.

Quite a lot of the desired functionality could be provided by a messaging service. Such a service will be considered as a part of the implementation of the information services. The extra features of the information service described here include the ability to issue queries across all the information, and to issue queries over historical information. A messaging service just delivers messages.

Any information can be monitored provided it carries a time-stamp. The mechanisms to move the information around are the same. What makes monitoring systems distinctive is normally the GUIs that are provided to visualise time-sequenced data and to highlight problems. These GUIs are simply clients of the information services.

Some information of interest changes rapidly and some much more slowly. However, even with the slowly changing information, it is often necessary to know quickly if it does change. Publishing information that is only changing infrequently, along with rapidly changing information is inefficient. This requires thought when designing schemas. It is better to treat the information as two or more entities with one to one relationships between them at any one time, rather than trying to bundle together slowly and rapidly changing quantities.

The Services to be provided are:

- Factory services
  - ProducerFactory service

  - ConsumerFactory service

- Producer services
  - PrimaryProducer service
  - SecondaryProducer service
  - OnDemandProducer service
- Consumer service

7.1.1 Other Services Used

VOMS is able to augment a user's certificate with information about groups and roles within a VO. The use of VOMS, or a similar service, facilitates the expression of rules to specify the desired granularity of access.

7.1.2 Producer services

The Producer services are used to publish information. A Producer is created by passing properties to a ProducerFactory service, which creates a Producer and returns a reference so you can interact with it. There are three main types of Producer:

- Primary
- Secondary
- On-demand

All behave in different ways, as described below, but have some common features.

Users of Producers declare tables to advertise the type of information that will be made available. The user can also specify a predicate (SQL `WHERE` clause) that defines the precise subset of the global table that will be published. In a Grid environment it is not common for all information contributing to one table to be published from a single source.

The user introduces new data into the information system by inserting it into a Primary Producer. Primary Producers are the initial source of the data. They may be thought of as having two internal stores, one for latest queries and one for other queries. Data are retained with the latest store for at least the duration of the minimum retention period, which must be set for each published tuple.

Producers may be implemented using different persistency technologies to store and organise their internal stores according to need. These include memory for speed, file based for recovery after a system crash and RDBMS to support both system recover and efficient joins.

Primary Producers transmit tuples to all listening Consumers, including those within SecondaryProducers. A Secondary Producer is used to aggregate streams of data and often at the same time make them persistent. They do this by setting up a Consumer to retrieve data for each declared table and republishing this through a Primary Producer. Secondary Producers are created by specifying persistency attributes of the Primary Producer that will be used to republish the data. As the user cannot directly publish new data using a Secondary Producer, it has no insert operation. The Secondary Producer has several uses:

1. It collects and maintains information in one place. This avoids querying Producers individually or accessing remote information sources.
2. It can be set up to record historical data and thereby act as an archiver of information.

3. It offers an alternative to the mediator for joins over tables held in different places.

An OnDemandProducer interacts with a user-supplied plug-in that returns data in response to an SQL query. It is used when the cost of publishing all the data would be too high compared to retrieving data only when it is requested by a Consumer. The plug-in can be rather complex to construct, and some limitations on the SQL it can accept must probably be imposed.

7.1.3 CONSUMER SERVICE

The Consumer Service is used to obtain data published by one of the Producer services. A Consumer handles a single query, expressed as an SQL SELECT statement. The Consumer also identifies how the query is executed, referred to as the query type. Depending upon this query type, the Consumer will run its query in one of two ways. The first is called a Continuous query and the second is referred to as a One-Time query. Continuous queries stream data from Producers. When a new tuple is published, the tuple is copied to all interested Consumers. This approach allows a Consumer to keep up-to-date with all Producer events. Conversely, one-time queries involve a single request/response for the Consumer to get information from a Producer. One time queries may in turn be either Historical or Latest; a Historical query has access to all the information and the Latest query only considers tuples with the most recent time-stamp for a table's primary (but without time-stamp) key. For all query types, the user can specify how far back in time to start. In addition there are a pair of retention periods to consider. One retention period is associated with history and continuous queries and is a property of the producer of that table. This is necessary to control the amount of historical data stored by a producer. The latest retention period is associated with the tuple and is not changed as data flows through the system. Latest queries never return tuples which are older than their latest retention period.

For all queries, retrieved tuples are stored within a Consumer buffer, managed by the Consumer service. The Consumer can extract buffered data by invoking the various pop operations available in the Consumer API.

Normally the Consumer uses mediator functionality within the Registry, to find the set of Producers to answer a query, however a user may direct a Consumer to a particular set of Producers if so wished.

7.1.4 SECURITY

Authorization rules, which are local to a VO, will define what actions an individual (certificate holder) may carry out. This includes the ability to publish information (via a Producer), to query (via a Consumer) or to discover what Producers exist. The authorization rules are specified when a table description is first defined in the schema. The authorization rules are defined in a TableAuthorization object that is passed into the createTable method. This holds a set of rules, each one a pair: (View : AllowedCredentials). Each View defines a view on a table in the form of a SELECT statement. If you match the allowed credentials you will have read access to the data defined in that view. It is possible that your credentials match two rules in which case you will be able to see the union of the two views. If you issue a query to see data you are not allowed to see, you will only be returned results derived from the data you are allowed to view. Both the View and the AllowedCredentials are parameterised in terms of VOMS attributes.

Authorization rules are local to a VO because the information system will give different views to each VO, with each VO having its own registry and schema. This is both to meet user requirements and to permit scalability.

In addition a means will be provided to allow users who are not members of a VO to have appropriate access to the system.

7.2 JOB MONITORING

There are two approaches to Job Monitoring i.e. allowing the detailed monitoring of what a job is doing. The code can be instrumented directly to publish what it is doing or a wrapper script can parse the outputs and publish information based on patterns in the output. Logically these are the same; it just depends on how the "job" is defined. The job simply wants to announce that something has happened, without being aware of whether or not anything is listening.

Within a Java application, the natural way to this this is with the java logging service or log4j[18]. Following the success of log4j, facilities supporting applications in other languages have been developed and now we see a new Logging Services[19] activity within Apache which "*is intended to provide cross-language logging services for purposes of application debugging and auditing.*" A tool, Chainsaw[17], is also part of the Apache logging service project. This is able to pick up logs and visualise them.

To make this work in a Grid environment one can use the producer and consumers of the information and monitoring service to provide transport across firewalls. This could either be achieved by writing a service on top of the information and monitoring service or wrappers could be provided on top of the producer API. The two solutions give the same appearance to the user. The first solution would be the one to adopt, following the current style of aggregation of services, except that it would be more efficient to follow the second route.

From an application point of view, messages (normally unstructured) are published using the normal logging API which publishes the data via the information system. The set of attributes which can be published will follow the log4j specification. As indicated, log4j messages are unstructured. If structure is required then the information and monitoring service should be used directly.

Facilities to make the data available to Chainsaw (or equivalent) will also be provided for visualisation.

7.2.1 OTHER SERVICES USED

Information and Monitoring will be used to publish the logging information.

7.2.2 SERVICES

As explained above, it is not yet clear whether to achieve the functionality by wrapping the Information and Monitoring API or to use service aggregation. In either case the user simply uses a standard logging API.

7.2.3 SECURITY

The underlying information system must protect job information from those not permitted to access it. The Authorization cannot be specified via the API as it is not part of the existing logging APIs, so instead the information will be passed directly to the component publishing the information into the Grid.

8 JOB MANAGEMENT SERVICES

8.1 ACCOUNTING

The accounting service accumulates information about the usage of Grid resources by the users and by groups of users, including Virtual Organisations as groups of users.

This information allows preparation of statistical reports, to track resource usage for individual users, to discover abuses and to help avoid them. Accounting information could be used to charge users for the Grid resources they have utilised.

The information available from the accounting service can also be used to implement submission policies based on user quotas or on resource usage (fair share). In principle it also allows the creation of a real exchange market for the Grid resources and services. The subsequent economic competition should result in market equilibrium, thereby promoting load balancing on the Grid.

Among the services from other Grid projects that can be compared with the EGEE Accounting Service it is worth mentioning Nimrod-G [4] and the EDG DGAS [39].

8.1.1 RESOURCE METERING

Figure 4 represents the actors involved in the "Grid accounting" schema. The key elements are the Grid resources and the Grid services. In order for an accounting system to work it is necessary to have reliable information about the Grid resource usage, thus each Grid resource or Grid service needs to be instrumented with dedicated sensors in order to measure the usage of the resource or service. This step of the process is the "resource metering" activity. The number of different type of resources is, in principle, very high. It is therefore necessary to foresee many different types of base metering sensors, that in the figure are shown as "Dedicated resource metering". Each type of resource will have its dedicated sensor. For example, different types of Local Resource Management Systems (LRMS) in a CE need different sensors and the metering infrastructure will also be different for a disk based Storage Element than for a tape library. It can also happen that there will be different implementations of a metering info provider for the same resource type, to preserve the freedom of the resource owner to choose one of the many available implementations, or to create a new one.

To allow the operation of the accounting layer, any distributed metering system has to report a basic set of information:

- A globally unique identifier of the user job or service request.
- A globally unique identifier of the user himself.
- A globally unique identifier of the Grid resource or service requested.
- User Accounting coordinates (see below).
- Resource Accounting coordinates (see below).

In order for the accounting system to deal with these varied sources of information a "metering abstraction layer" service is needed. Its purpose is to translate the various Usage Records' reporting protocols used by the metering sensors into an appropriate Usage Records format that the accounting system can understand.

8.1.2 ACCOUNTING SERVICE

Once the raw usage records coming from the many different info providers are translated, they can be forwarded to the Grid Accounting system. The accounting system is responsible for archiving these Usage Records and providing querying functionalities. We identify the following basic query functionality:

- Report aggregate or detailed Usage Records for a User.
- Report aggregate or detailed Usage Records for a resource.

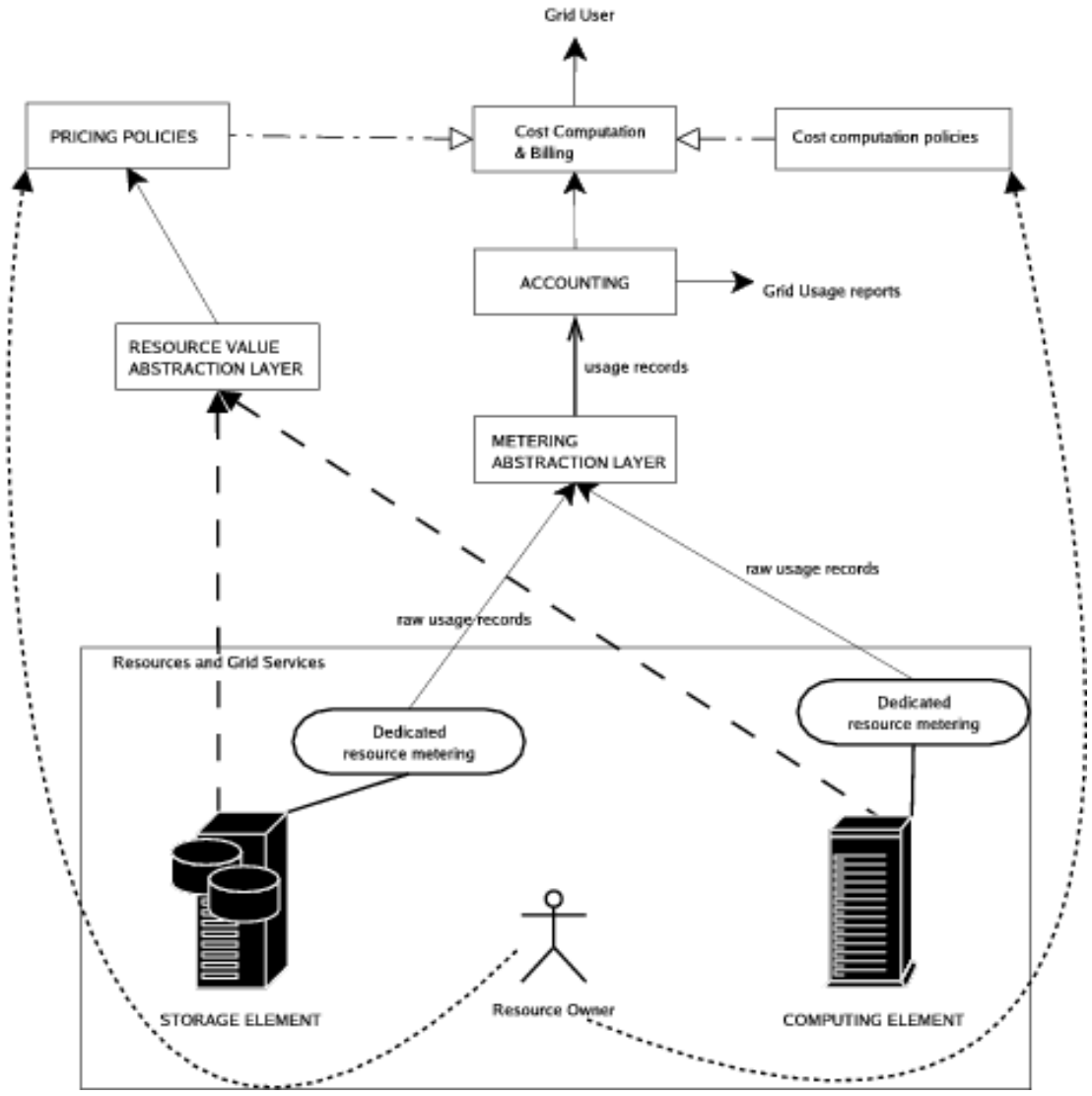

Figure 4: Fundamental actors in the provision of an accounting service.

- Report aggregate or detailed Usage Records for a group of users (including VOs).

As for every Grid service, it is important for the accounting to be scalable, so its architecture must be distributed. Many local accounting infrastructures are based upon a central database where the Usage Records for each user are kept: this is clearly not feasible in a Grid environment.

To satisfy this requirement, many independent accounting servers are foreseen. Both Grid users and resources have an "account" on one of these servers and all the information concerning their resource usage is kept on that server. There is no limit on the number of the servers. Upon each Grid service request, the user specifies his accounting coordinates (very close in concept to banking coordinates). Each Grid service or resource also specifies its accounting coordinates (these should be published on the resource information provider as well). Then, on completion of a job, or service request, the relevant usage records are pushed to both the user and the resource accounting servers. Practically we require the user to specify accounting coordinates as a parameter when submitting a job. This parameter then needs to be propagated upon each subsequent Grid service request originating from the job.

A practical example of how to partition users over the accounting servers is to have one accounting server per VO (or more for large VOs). In this way the accounting coordinates specified by the user will identify the VO related account that the user intends to charge for a given job.

8.1.3 Cost computation and Billing

The next layer in a general purpose accounting system is the "billing" for resource usage. The infrastructure described so far is absolutely independent from the billing infrastructure and can, in principle, be used stand-alone.

When billing is needed, resources need to be priced, so that a cost can be assigned to the users' activities.

The price for a Grid resource should be assigned according to its real value, that is, according to the overall performance it offers to the users. The information used to estimate the resource value may change depending upon the resource purpose. The information needed for pricing a computing element is intrinsically different from that of a tape library.

Pricing information may also come from providers in different formats, so another abstraction layer can be introduced to translate such information into a format that the upper pricing layer can understand.

Once the relevant information about the resources is available, a service responsible for assigning the prices uses it, along with a set of predefined pricing policies to calculate the resource prices. It is important for the resources owners to be able to interact with the price calculation by defining the pricing policies. In practice it is the resource owner who decides the algorithm used to compute the price, even if some type of limitations can be imposed at an upper ("Grid" or "VO" management) level.

¿From an architectural point of view the pricing service has the same requirements of scalability as the accounting service, so it is expected that Grid resources will register themselves with a pricing service and publish the service address.

The last step is the computation of the cost for a user job, or service request, according to the resource price, the amount of resources used (reported by the accounting service as Usage Records) and the cost computation policies.

The cost computation policies are a set of rules imposed by each resource owner that specify how to compute the cost that has to be billed to the user. Usually it will be a formula that computes the job cost starting from the resource price and the job usage records. As an example the usage of a Grid service can be billed on a per-transaction basis instead of usage metering. In this scenario the service owner will bill a fixed amount of credits for the service usage: this can be represented at the cost computation policy level. Another possible scenario is a resource owner who wants to sell resources at a discounted rate (or

offer them free) to a group of users, but wants a standard rate to be applied to the other users. Resources prices should be publicly available in order for users to choose where to submit jobs. Additional pricing policies such as discounts arranged with a particular VO could be confidential. Such policies are specified at resource level.

It has to be stressed that even if the resource is free of charge, or users pay for the usage on a per-transaction basis, it is important for users to preserve the ability to retrieve the Usage Records corresponding to their work in order for them to check, for example with a properly instrumented work request, whether the metering infrastructure is reliable.

Once the amount to be charged is decided, it is communicated to the accounting level where a bank service implements the virtual payment between the user Grid bank service and the resource one. It is the accounting layer itself that manages the economic accounts for users and resources.

A design aspect that needs to be stressed is that the security of the communication involving accounting information is essential. The Usage Records treated by the accounting system can be used by the Grid site managers and service providers to charge the user for their resource usage or, simply, to track abuses. This implies that this information must be trustworthy and it must be impossible for the Grid user to repudiate it. A practical way to achieve this goal is to mutually authenticate every network communication involving accounting information: the user authenticates to both the service provider and the accounting system. The Monitoring System sends the Usage Records to the accounting on behalf of the user, for example using delegated credentials. At the same time the user requires the remote peer to authenticate itself, so that he can be sure that his Usage Records are sent to the desired Accounting service and not to a malicious one. Another key aspect of accounting information is confidentiality. It is clear that the accounting service treats sensitive information, and users (or the law) may require such information not to be public. Communications should therefore be encrypted. Also, for confidentiality reasons, a proper authorization policy must be enforced for access to the accounting database. The Accounting records should be accessible only by the owner or by trusted administrators. A typical ACL would allow users to access their own records but not those of other users, while allowing VO administrators to access the records belonging to users of that VO.

8.2 COMPUTING ELEMENT

The Computing Element (CE) is the service representing a computing resource.

Its main functionality is job management (job submission, job control, etc.), but it must also provide other capabilities, such as the provision of information about its characteristics and status.

Comparable services from other Grid projects include: the EDG CE, the Alien CE and the Globus GRAM.

The CE, exposing a Web Service interface, may be used by a generic client: an end-user interacting directly with the Computing Element, or the Workload Manager, which submits a given job to an appropriate CE found by a matchmaking process (see Section 8.3).

A CE refers to a set, or *cluster* of computational resources, managed by a LRMS. This cluster can encompass resources.that are heterogeneous in their hardware and software configuration. When a CE encompasses heterogeneous resources, it is not sufficient to let the underlying LRMS dispatch jobs to any worker nodes. Instead, when a job has been submitted to a CE, the underlying resource management system must be instructed to submit the job to a resource matching the requirements specified by the user.

The interface with the underlying LRMS must be very well specified (possibly according to existing standards), to ease the integration of new resource management systems (even by third party entities) as needed. The definition and provision of common interfaces to different resource management systems is still an open issue, but there are proposed recommendations currently under discussion (such as the

Distributed Resource Management Architecture API, DRMAA, currently discussed within the GGF), which are likely to be adopted when they become consolidated standards.

8.2.1 JOB MANAGEMENT FUNCTIONALITY

The main functionality that the Computing Element has to provide is job management. It must provide facilities:

- To run jobs (which includes also the staging of all the required files).
  Characteristics and requirements of jobs that must be executed are specified relying on the same Job Description Language (JDL) [36], used within the whole Workload Management System.
- To get an assessment of the foreseen "quality of service" for a given job to be submitted. This reports, first of all, if there are resources matching the requirements and available according to the local policies. It then might provide an estimation of the local queue traversal time, that is the time elapsed since the job entered the queue of the LRMS until it starts execution.
- To cancel previously submitted jobs.
- To suspend and then resume jobs, if the LRMS allows these operations.
- To send signals to jobs.
- To get the status of some specified jobs, or of all the active jobs "belonging" to the user issuing the request.
- To be notified on job status, for example when a job changes its status or when a certain status is reached.

For job submission, the CE will be able to work in *push model* (where the job is pushed to a CE for its execution) or *pull model* (where the CE is asking the Workload Management Service for jobs).

When a job is pushed to a CE, it gets accepted only if there are resources matching the requirements specified by the user, and which are usable according to the local policies set by the local administrator. The jobs gets then dispatched to a worker node matching all these constraints.

In the pull model, instead, when a CE is willing to receive a job (according to policies specified by the local administrator, e.g. when the CE local queue is empty or it is getting empty) it requests a job from a known Workload Management Service. This notification request must include the characteristics and the policies applied to the available resources, so that this information can be used by the Workload Management Service to select a suitable job to be executed on the considered resource.

Various scheduling mechanisms within the pull model must be investigated to determine which provide the best performance in various situations when a CE willing to receive a job for execution, has to refer to multiple Workload Management Services. Possible mechanisms include:

- The CE requests a job from all known Workload Management Services. If two or more Workload Management Services offer a job, only the first one to arrive is accepted by the CE, while the others are refused.

- The CE requests a job from just one Workload Management Service. The CE then gets ready to accept a job from this Workload Management Service. If the contacted Workload Management Service has no job to offer within a certain time frame, another Workload Management Service is notified. Such a mechanism would allow supporting priorities on resource usage: a CE belonging to a certain VO would contact first a Workload Management Service referring to that VO, and only if it does not have jobs to be executed, the Workload Management Services of other VOs are notified, according to policies defined by the owner of the resource.

8.2.2 OTHER FUNCTIONALITY

Besides job management capabilities, a CE must also provide information describing itself.

In the push model this information is published in the information Service, and it is used by the match-making engine (see Section 8.3) which matches available resources to queued jobs. In the pull model the CE information is embedded in the "CE availability" message, which is sent by the CE to a Workload Management Service. The matchmaker then uses this information to find a suitable job for the CE.

The information that each CE should provide will include:

- the characteristics of the CE (e.g. the types and numbers of existing resources, their hardware and software configurations, etc.);
- the status of the CE (e.g. the number of in use and available resources, the number of running and pending jobs, etc.);
- the policies enforced on the CE resources (e.g. the list of users and/or VOs authorized to run jobs on the resources of the CE, etc.).

As described in Section 8.1, each CE is also responsible for the *Grid accounting resource metering*, that is, it must measure user activities on the CE resources, providing resource usage information. This information, after having been properly translated in an appropriate format, is then forwarded to the Grid Accounting System.

8.2.3 INTERNAL CE ARCHITECTURE

Figure 5 represents the architecture of the Computing Element.

Considering a job submission, the *Computing Element Acceptance* (CEA) service, exposing a Web Service interface, represents the entry point for submitting jobs to the resources of the CE.

The CEA includes the functionality of a *Site Gatekeeper*, responsible for the mapping between Grid users and local users, and for checking if the job can be accepted according to the configuration options that could have been set to limit the load caused by job processing.

After having checked that the considered job can be executed in the CE (that is there are resources matching the constraints specified in the job JDL expression and which can be used according to the local policies), the CEA updates the User Context (UC), a data structure holding information about the user and the active jobs she/he owns on the CE, accordingly.

Then the job is forwarded to the *Job Controller* (JC). The JC is in charge of submitting the job to the underlying LRMS and controlling its execution. The client interacts with the JC (also exposing a Web Service interface) to control jobs: to get their status, to suspend them, to kill them, etc. The JC relies on the information stored in the user's UC when serving a request on that user's job.

The *Monitor* (MON) service deals with notifications. It can be customized in particular to:

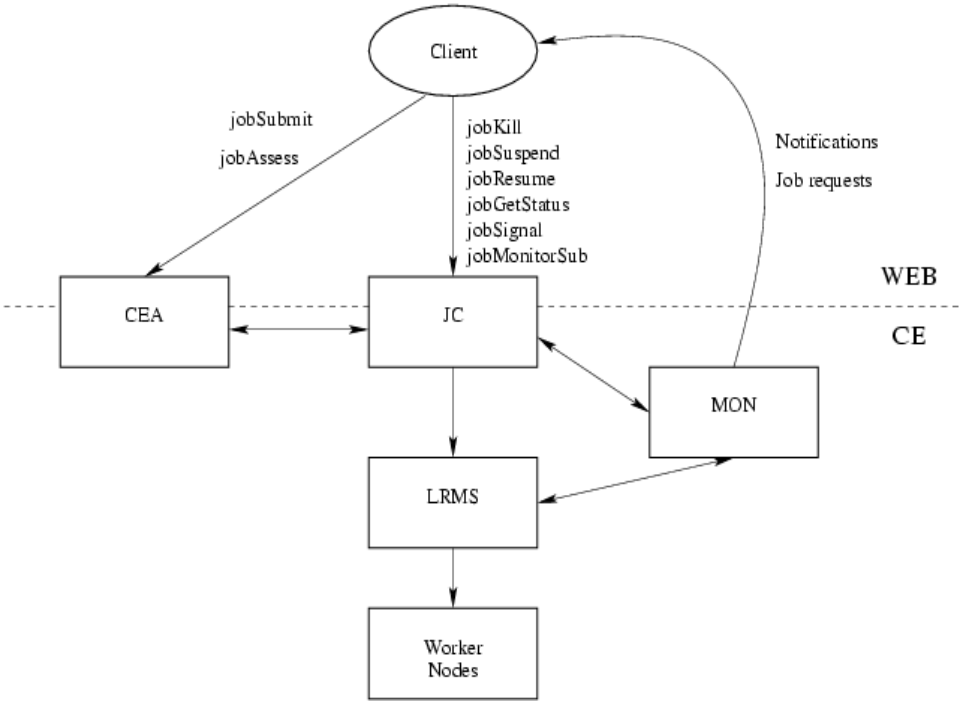

Figure 5: Architecture of the Computing Element.

- asynchronously notify users on job status events, according to policies specified by users (e.g. when a job changes its status, when a job reaches a certain status, etc.). The jobs to be monitored and the type of notifications to support are stored in the user's UCs;
- notify about the CE characteristics and status. In particular, for a CE working in pull mode, this service is used to request jobs to the Workload Management Service.

8.2.4 POLICY DEFINITION AND ENFORCEMENT

As already mentioned, the resources belonging to Computing Elements must be used according to some specified policies, set by the administrators of these resources. Examples of policies include the list of users or VOs allowed to use the resources of the CE, the share of the CE resources available to the users belonging to a certain VO, etc.

All the defined policies have to be taken into account when choosing the CE where to submit user jobs, and when selecting the computational resources within a single CE to dispatch jobs to (in case the computational resources within a single CE are managed by different policies).

In order to accomplish this task, the CE will make use of the policy combination and authorization framework described in Section 5.2.2.

8.3 WORKLOAD MANAGEMENT

The Workload Management System (WMS) comprises a set of Grid middleware components responsible for the distribution and management of tasks across Grid resources, in such a way that applications are conveniently, efficiently and effectively executed.

Comparable services from other grid projects are, among others, the EDG WMS [29, 30], Condor and the Eurogrid-Unicore resource broker.

The specific kind of tasks that request computation are usually referred to as "jobs". In the WMS, the scope of tasks needs to be broadened to take into account other kinds of resources, such as storage or network capacity. This change of definition is mainly due to the move from batch-like activity to applications with more demanding requirements in areas like data access or interactivity, both with the user and with other tasks. The WMS will broaden its scope accordingly.

8.3.1 FUNCTIONALITY

The core component of the Workload Management System is the Workload Manager (WM), whose purpose is to accept and satisfy requests for job management coming from its clients. The other fundamental component is the Job Logging and Bookkeeping Service, which is described below.

For a computation job there are two main types of request: submission and cancellation (the status request is managed by the Logging and Bookkeeping Service).

In particular the meaning of the submission request is to pass the responsibility of the job to the WM. The WM will then pass the job to an appropriate CE for execution, taking into account the requirements and the preferences expressed in the job description. The decision of which resource should be used is the outcome of a *matchmaking* process between submission requests and available resources. The availability of resources for a particular task depends not only on the state of the resources, but also on the utilisation policies that the resource administrators and/or the administrator of the VO the user belongs to have put in place (see Section 8.2.4).

Describing the functionality of the WM as submitting a classad-based [42] job request, being able to control it (cancel, possibly send signals to it, put it on hold and release it), and get its status, shows that the WM has an interface that is very similar to the one of the CE (see Section 8.2). Whether this can be exactly the same (which is among our goals) depends largely on finding a suitable solution to issues for the CE such as maintaining a consistent and reliable view of the job request as it travels through the system and making sure that no components in the CE can cause job requests to be lost.

8.3.2 SCHEDULING POLICIES

A WM can adopt a more or less eager or lazy policy in order to schedule a job. At one extreme, eager scheduling dictates that a job is bound to a resource as soon as possible and, once the decision has been taken, the job is passed to the selected resource for execution, where, very likely, it will end up in a queue. At the other extreme, lazy scheduling foresees that the job is held by the WM until a resource becomes available, at which point that resource is matched against the submitted jobs and the job that fits best is passed to the resource for immediate execution. Varying degrees of eagerness (or laziness) are applicable.

At match-making level the main difference between the two extremes is that eager scheduling implies matching a job against multiple resources, whereas lazy scheduling implies matching a resource against multiple jobs.

The WM internal architecture will accommodate application of the different policies, implemented as easily interchangeable plugins, depending first of all on the requirements and preferences expressed in the job description, but also on the overall state of the Grid, according to appropriate heuristics. Such knowledge can only come from proper investigation (including the evaluation of relevant metrics, covering both resource utilisation and user satisfaction), with the purpose to understand strengths and weaknesses of the different scheduling policies in different scenarios.

8.3.3 THE INFORMATION SUPERMARKET

The mechanism that allows the flexible application of different policies is the decoupling between the collection of information concerning resources and its use. The component that implements this mechanism is dubbed *Information Supermarket* (ISM) and represents one of the most notable improvements in the WM as inherited from the EU DataGrid (EDG) project.

The ISM basically consists of a repository of resource information that is available in read only mode to the matchmaking engine and whose update is the result of either the arrival of notifications or active polling of resources or some arbitrary combination of both. Moreover the ISM can be configured so that certain notifications can trigger the matchmaking engine. These functionalities will not only improve the modularity of the software, but will also support the implementation of lazy scheduling policies.

8.3.4 THE TASK QUEUE

The second most notable improvement in the WM internal design is the possibility to keep a submission request for a while if no resources are immediately available that match the job requirements. This technique is used, among others, by the AliEn [44, 6] and Condor systems. Non-matching requests will be retried either periodically (in an eager scheduling approach) or as soon as notifications of available resources appear in the ISM (in a lazy scheduling approach). Alternatively such situations could only lead to an immediate abort of the job for lack of a matching resource.

The component that implements this feature is dubbed *Task Queue* (TQ) and, as for the ISM, provides a necessary mechanism for the support of lazy scheduling policies.

8.3.5 JOB LOGGING AND BOOKKEEPING

The Logging and Bookkeeping service (L&B) tracks jobs in terms of *events*—important points of job life, e.g. submission, finding a matching CE, starting execution etc.—gathered from various WMS components as well as CEs (all those have to be instrumented with L&B calls). The events are passed to a physically close component of the L&B infrastructure (*locallogger*) in order to avoid network problems. This component stores them in a local disk file and takes over the responsibility to deliver them further.

The destination of an event is one of *bookkeeping servers* (assigned statically to a job upon its submission). The server processes the incoming events to give a higher level view on the job states (e.g. *Submitted, Running, Done*) which also contain various recorded attributes (e.g. JDL, destination CE name, job exit code, etc.). Retrieval of both job states and raw events is available via legacy (EDG) and WS querying interfaces.

Besides querying for the job state actively, the user may also register for receiving notifications on particular job state changes (e.g. when a job terminates). The notifications are delivered using an appropriate infrastructure.

8.3.6 THE OVERALL ARCHITECTURE

Figure 6 shows the overall architecture of the Workload Manager, together with the interactions with external entities. Among these the most coupled with the WM is the Logging and Bookkeeping Service, which keeps events generated by different components as a job traverses them. Such events contribute to the generation of the status of a job. Other entities are the Information System (see Section 7), used, for example, to fill the Information Supermarket, the Data Management services (see Section 9), assisting the WM when the scheduling involves knowledge concerning location of data on the Grid and the Access Policies infrastructure (see Section 8.2.4).

Both the WM and the other services are expected to offer a Web Service interface.

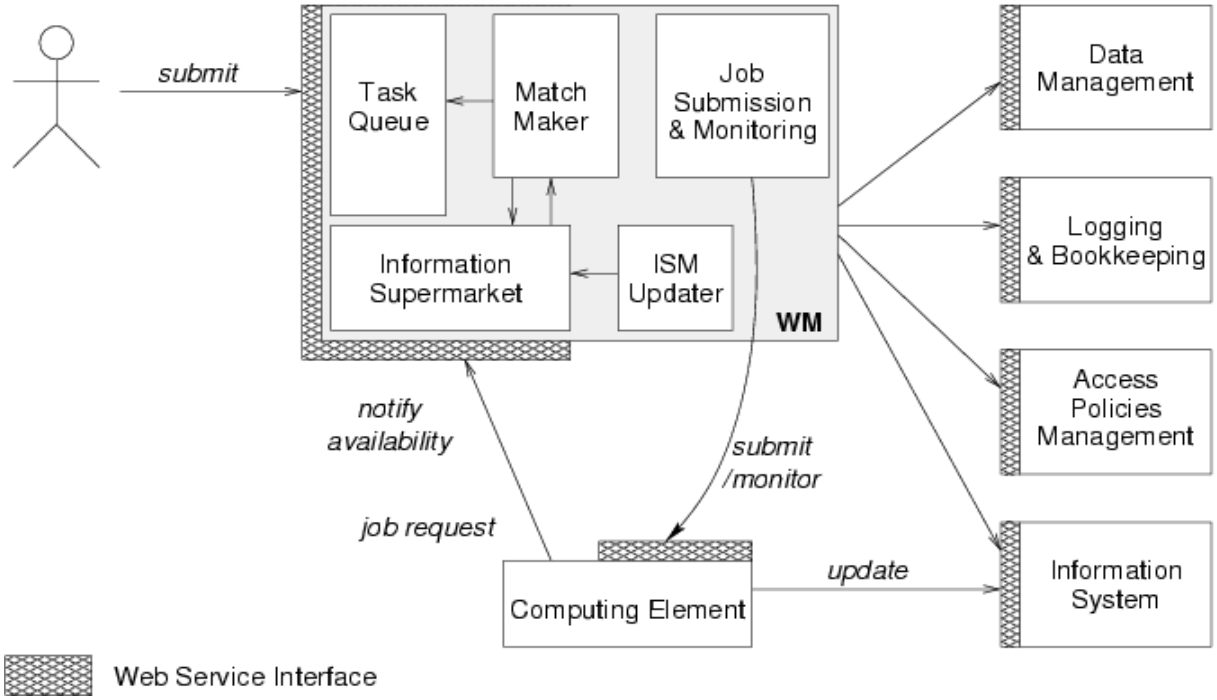

Figure 6: Internal architecture of the Workload Manager.

8.4 JOB PROVENANCE

8.4.1 PURPOSE, EXPECTED USAGE, AND LIMITATIONS

The purpose of the Job Provenance service (JP) is keeping track of the definition of submitted jobs, execution conditions and environment, and important points of the job life cycle for a long period (months to years). Those data can be used for debugging, post-mortem analysis, comparison of job execution within an evolving environment, as well as assisted re-execution of jobs. Only data of completed (either successful or failed) jobs are handled; tracking jobs during their active life is the task of L&B and Job Monitoring services described elsewhere in this document.

In general, gathered data are stored (i.e. copied) within the JP storage in order to really conserve a partial snapshot of the Grid environment when the job was executed, independently of changes of other Grid services. Obviously there are practical limitations of the extent to which it is feasible to record the entire job execution environment. (In the ideal case this would encompass a snapshot of the entire Grid!) We restrict the recorded data to those that are processed or somehow affect processing of the Workload Management and Computing Element services. On the other hand, snapshots of the state of other Grid services are not done, namely queries to the Data Catalog and their results are not stored, as well as contents of data files downloaded from and uploaded to Storage Elements—only references to those are recorded if required.

8.4.2 ENCOMPASSED DATA AND THEIR SOURCES

Data required by the Job Provenance service are gathered by various Grid middleware components. Depending on the design of a particular component the data are either “pulled” by the JP from the component, or the component has to be instrumented to “push” the data towards JP.

Certain minimal records of a job (see the job life log below) are always stored in order not to lose the job completely. However, the complete JP data may grow rather large, and it strongly depends on the

specific context whether it makes sense to gather the complete data or only a subset. In general, policies at several levels (at least WM and CE) define which data should and could be gathered, i.e. enforce what is mandatory and what is prohibited, allowing the user to specify preferences within those bounds. In general, those policies are specified by "owners" of the resources (WM, CE).

The following data are gathered:

Job life log taken over from the L&B database (see Sect. 8.3.5) after job completion. Among other information useful mainly for debugging and detailed analysis of job execution it contains the complete definition of the job (in terms of the submitted JDL), various timestamps (e.g. when the job was submitted, matched for a particular CE, started and finished execution), information on the chosen CE (or more of them, if the job was resubmitted), various ID's assigned to the job (Condor, Globus, LRMS, ...), and the result of execution including the return code.

Another important information available in L&B are the user-defined annotations ("tags") which can be specified either statically upon job submission, during its execution, or even overridden after the job terminates.

All the information is copied out from L&B and can be purged eventually.

Executable file(s) are provided by the user upon submission. In the case of high-throughput jobs the executable is the same for many (thousands or more) jobs. Therefore for efficient storage of the executable, JP has to allow sharing a stored executable among multiple jobs.

The executable is managed by the submission interface (aka Network Server) of the Workload Management service. This service has to be instrumented to cooperate with the JP storage.

Input/Output sandbox The input sandbox of a job contains miscellaneous files required for execution, and the output sandbox may contain various output files, e.g. debug logs, or even a core file in the case of crash.

By default only the input sandbox is stored—the output is assumed to be reproducible by the job. The user (or a WM/CE policy) may trigger storing the output sandbox as well. For both input and output, either the whole sandbox or enumerated files only may be stored.

The sandbox files to be stored by JP are specified upon job submission. If required, mechanisms for adjusting the list (both adding and removing files) could be provided.

Staging the sandboxes in and out is managed by the Network Server too. Therefore the same instrumentation applies.

Input/output files In contrast to the files contained in the sandboxes these are downloaded from and uploaded to Storage Elements. JP does not copy those files but may record references to them.

Execution environment comprises of operating system and installed software versions, and specific configuration information (including setting of environment variables) of the particular worker node of the CE where the job has been executed.

We assume that the primary source of the most up-to-date information of this type is the worker node itself. Therefore the script which wraps the invocation of the job executable will be instrumented to log the required information. However, certain information can be also retrieved from CE configuration management service.

The potential extent of information in this category is huge, with varying meaningfulness for particular purpose. Therefore a rigid generic approach is inappropriate. Instead we let the user (or policy) choose what should be recorded. JP provides a predefined set of typical data items to gather (e.g. output of "rpm -qa", enumerated environment variables, ...), as well as the facility to execute a custom plugin script to collect specific data.

Custom data may be provided by the user upon submission, gathered during job execution (via plugin script called by the job wrapper), or even after the job termination.

The described items form a complete set of data that can be gathered by the JP service. However, depending on conditions and requirements which could not be estimated at the design time, some of those data may be irrelevant for a certain type of jobs. As this can result in wasting considerable storage resources we allow restricting the complete set by setting user preferences and/or applying site policy of a particular instance of the JP service.

8.4.3 SERVICE COMPONENTS

Primary storage servers keep the JP data in the most compact and economic form. The access key to the data is job ID, and we assume always accessing data on a job as a whole. All the data related to a single job form a *Job Record* containing the input and output sandboxes, executable (or a reference to it), and the L&B job dump. The file is named after the job ID which is supposed to be never recyclable hence no collisions should occur.

In order to allow access to Job Records without the explicit knowledge of job ID's, certain basic metadata are managed by the primary storage. The exact set of attributes has to be clarified, we are considering job owner DN, submission time, and virtual organisation.

Primary storage provides public interfaces for data storing, retrieval based on basic metadata, and registration of Index servers for incremental feed (see below).

Index servers provide a limited data mining capability on the JP data. Each index server is configured by its administrator to support a set of *queryable attributes* chosen from various L&B system attributes like job submission, start, and termination time, CE name, exit code (more or less the set supported by indices of the L&B server), arbitrary L&B user tags, as well as non-L&B attributes related to the other JP data (E.g. input/output sandbox file names, environment variable settings. For the sake of coherent processing those are transformed, solely for the purpose of indexing, into L&B-like pseudo events.).

A result of a query is the set of matching job IDs and corresponding URLs (`gridftp://`, `lfn://`, ...) according to which the data from the JP storage can be retrieved.

There are two modes of feeding data into the index server:

- *batch*—the server is populated from scratch with data retrieved from the JP storage, typically jobs submitted in a given time interval.
- *incremental*—whenever data on a new job are stored to the JP storage server, those have to be propagated to the index server as well so that further queries are able to find this job.

In general, the relationship between the index and storage JP servers is many-to-many; several index servers (of different index configuration) can be populated with data from a single storage, the data may even cover different time intervals. On the other hand, a single index server may combine data from multiple JP storage servers.

In contrast to the permanent primary storage the index servers are volatile. They may be created, populated with data, and destroyed independently of the primary storage.

The querying interface will be exposed as a web-service. The index server will be also interfaced to the information infrastructure (R-GMA) as an on-demand producer, thus allowing complex distributed queries to be performed over multiple JP index servers.

Analysis support tools Besides the public WS interface to both the primary storage and index server JP provides tools for managing the index servers, e.g. purging and populating them with data coming from the primary storage, as well as manipulating the set of queryable attributes of a particular index server.

Job resubmission support tools JP data contain enough information to re-create the execution environment of a particular job. However, the user may require varying fidelity of the environment reproduction, e.g. preserving the same environment settings but using an upgraded version of a particular library. Therefore support for fine-tuning the specification of the resubmitted job has to be provided. However, this is foreseen as a specific support in existing user interfaces, using the JP service in turn.

8.4.4 SECURITY

Authentication All the communication involved in JP is authenticated using appropriate user or service credentials.

Encryption A lot of the gathered data may be considered sensitive, hence encryption of data transfers should be considered. However, this may generate a considerable overhead in the case of bulk file transfers, therefore fine grain control is required (e.g. encrypt input/output sandboxes but not executables).

Authorization

- *Storing to primary storage* is done by L&B, WM (sandbox management part) and CE eventually. Only credentials of configured trusted services are permitted.
- *Read access* to both primary storage and index servers. ACLs (including both user identities and VOMS groups) are inherited from L&B upon creation of the Job Record.

  If the primary storage back-end (SE service) allows, its access control mechanisms are exploited as well.
- *Modification of Job Records.* To be defined later if required. Also ACL manipulation will be considered.
- *Feeding data into index server.* Mutual trust between primary storage and index server is required. This is based on both primary storage and index servers' configurations.

8.5 PACKAGE MANAGER

A Package Management (PM) Service is a helper service that automates the process of installing,upgrading, configuring, and removing software packages from a shared area (software cache) on a grid site. This service represents an extension of a traditional package management system to a Grid and it should use one of the established package management systems as a back-end. Some well-known examples of such systems include:

- RPM, Red Hat's package manager, used not only by Red Hat Linux but by several other Linux distributions.
- dpkg/APT (used originally by Debian GNU/Linux, now ported to other systems).
- Portage, used by Gentoo Linux and inspired by the BSD ports system.
- The "ports tree" system used by FreeBSD, NetBSD,OpenBSD and the like.
- Pacman, package manager developed at Boston University and used by several Grid projects (International Virtual Data Toolkit - iVDGL, Grid3)

The software is distributed in packages, usually encapsulated into a single file. The file, as well as the software itself, contains metadata that describes the package's details, including its name, checksums, and dependencies on any other packages that it needs to work. It may also include information on how to configure the package for use and how to remove the package cleanly when it is no longer required. The package manager then uses this information to install, configure, and remove packages as requested by the user.

The PM Service is used by other services running in a resource provider environment (Computing Element and Job Wrapper) in order to construct the list of packages that are required to execute a job specified by the JDL. The information returned is an LFN or SURL. Based on that, the application (CE or JW) can retrieve the required files, install them in the package directory and make them available on the WN.

The PM Service operates in the context of a VO and understands and resolves possible dependencies between the package versions provided by the VO administrator. This service is not responsible for the maintenance and deployment of middleware or system software components.

The VO package administrator maintains the package cache at VO level consisting of a tarball and a metadata file per package. The tar file can contain the precompiled binaries for a given platform or the source files that can be compiled to produce executables. The associated package metadata file contains the build instructions for the source files, possible dependencies on other packages as well as the instructions on how to setup the runtime execution environment.

In a typical scenario, the VO package administrator creates the binary package caches for one or more computing platforms, verifies and possibly digitally signs them. These caches are then published and made available for download via the PM Service. On the execution site, a local instance of the PM Service will, on request from a CE or JW, fetch and install binary packages into the local package cache. This local package cache should reside on a file system managed by the PM Service assuring that unused old packages are removed if disk space is needed to install newer versions.

However, should the VO decide to manage only the source caches, the packages can be built by the local PM Service, using the build instructions contained in package metadata and installed in the local package cache. In addition, the existence of binaries can be advertised, thus minimising download of packages from multiple locations. In this way, the PM Service could maintain the hierarchy of package caches to assure scalability and provide a fail-over capability.

8.5.1 Other Services Used

Authentication, Authorization, File Catalog

8.5.2 Security

Access to VO packages should be controlled and possibly restricted and audited. The easiest way to achieve that is to treat the packages as any other File Catalog entry and to apply common Authentication, Authorization and Auditing mechanisms. The integrity of individual packages should be verified by appropriate checksums. The package metadata information (including checksum information) should be retrieved from a trusted and certified VO site, independently from the package itself.

9 DATA SERVICES

9.1 STORAGE ELEMENT

The Storage Element (SE) is responsible for saving/retrieving files to/from the local storage that can be anything from a disk to a mass storage system. It manages disk space for files and maintains the cache for temporary files.

The SE provides the following set of capabilities which will all be explained in detail in this section:

- Storage space for files. In an SE there is a certain amount of space available to store file-based data.
- Storage Resource Management interface. The SE has a standardised interface to manage the underlying storage resource. Currently this is mainly for staging files from and to disk from other media.
- Storage space management. The SE may provide a means to manage the available space through mechanisms such as quotas, file lifetime management, pinning, space reservation, etc. We do not oblige each SE to provide such functionality, but if the functionality is available it should be used. The SRM interface is being extended to provide this capability in a standardised manner.
- POSIX-like I/O access to files. The SE provides an interface that can be used to access the files directly through some supported protocol.
- File Transfer requests. Logically there is a File Transfer Service interface coupled to each SE, controlling all data influx into the SE from other SEs.

Storage Elements represent a resource that may have very different quality of service (QoS) characteristics between different instances, as well as different levels of portability; see Figure 7. The “portability” in this sense means ease of deployment, depending also on the actual storage technology. Storage Element middleware running on a tape archiver is considerably more complex than a thin layer of logic on top of a local hard-disk in user-space. In-between these two extremes there is anything from disk-arrays to DVD racks. The “QoS” axis refers to the “safeness” of the data being stored; high QoS storage means less probability of data loss. Such high-QoS systems have well-defined backup and restore mechanisms in place, sometimes even geographically dispersed mirrors (in case of a natural disaster wiping one of the copy including all local backups). As described below, low QoS storage would be an opportunistic storage where someone offers some space on a hard-disk for a limited time, with no backup strategy nor a commitment to keep the data.

Opportunistic, “tactical” Storage

At one extreme, the low QoS and high portability end, we have Storage Elements that have a very simple purpose: store grid files close to a computing element as long as they are needed (if possible). Such SEs can be viewed as file caches, temporary or even scratch space. Their value is in their flexibility. Since they do not provide high data safety, such SEs can be easily deployed, switched on or off according to the need of a given site or a given virtual organisation.

SEs are controlled by the sites and are subject to local policy. Having the concept of tactical SE allows a site to declare space in a local store in an opportunistic manner. It can provide storage that is currently unused by their local users and revoke it whenever necessary. This will make it attractive to sites to make resources available to grid users, knowing they can re-claim the resources whenever they want. Thus, jobs requiring local storage may be run at many more sites. Users are expected to keep only disposable data in such stores - meaning that it should not matter if the instance of the data is lost - because it can be

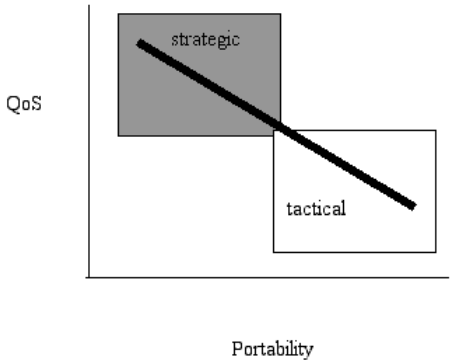

Figure 7: Space occupied by our different storage concepts in the space of QoS and Portability.

re-generated or re-copied from a master instance for example. Important data, master copies should not be kept in such storage (only at the user's own risk). If users generate new data at such stores, they should either register a master copy at a more long-term SE or be prepared to re-generate the data if necessary.

The tactical SEs may be hot-deployable and be alive only for a short period of time.

Permanent, "strategic" Storage

Such storage comes with a higher quality of service. Users may expect to be able to reliably retrieve their files from such storage at any later time. Such storage usually has an expensive managed MSS behind it. The administrative overhead is much larger than for opportunistic storage.

Wherever we use the term strategic Storage Element, we think of a permanent, reliable storage to store master copies or other important data.

EGEE middleware will provide both the strategic and the tactical SE with the appropriate interfaces (see below for details). These interfaces are the SRM interface and a POSIX-like I/O API. The API might have some differences between strategic and tactical SEs, most certainly in the interface semantics, but some operations may not exist on a tactical SE that exist on a strategic one (especially for the SRM interface) and vice versa. Basically the site deploying the SE will configure the SE as either tactical or strategic. The users should be aware at all times what kind of SE they use to store their data.

The actual middleware implementations for the SE will come in many cases from other projects, such as from DESY for dCache, CERN for Castor, SDSC for SRB[4], RAL for the ADS[5], Condor NeST, etc.

9.1.1 Other Services Used

The Storage Element is making use of the catalog services to resolve GUIDs and LFNs to SURLs and to check authorization for the files. The services needed to do this resolution should be deployed next to the SE. This minimises the probability of the SE files being inaccessible due to unavailability of the catalog.

The SE also makes use of the monitoring services to publish its state information, and of the security services for the purpose of authorization and accounting.

9.1.2 Services

There are two interfaces into the SE (see Figure 8). The first interface is the Storage Resource Management (SRM) interface. This interface allows the client to manage the storage space - to allocate space

[4]The San Diego Supercomputing Center's Storage Resource Broker
[5]Rutherford Appleton Laboratory's Atlas Data Store

for jobs, to prepare data to be retrieved through a certain protocol, etc. We foresee the evolution of that interface according to the proper SRM specifications, see [24]. We might extend this interface if we see the need, while keeping in sync with the SRM specification group and trying to work our changes that we feel necessary into the specification.

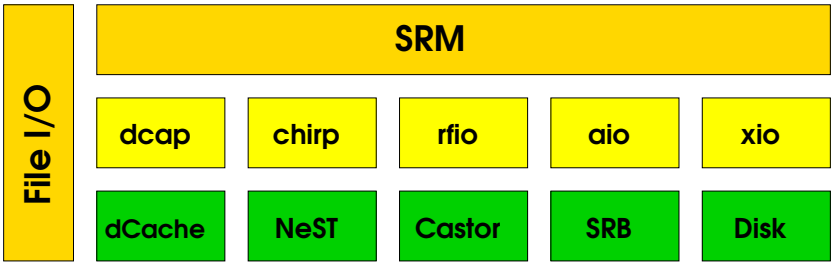

Figure 8: The SE modular breakdown. The SE has two external interfaces, the SRM and a POSIX-like file I/O interface. The SE may support a large number of protocols to access the files. The list in the picture is non-exhaustive.

The second interface that is exposed is a POSIX-like file I/O API. These SRM and I/O interfaces will make use of many third-party storage and protocol components, but exposing a uniform interface to the client should keep the client shielded from local peculiarities. All the different semantics of different storages should be exposed in the information system and through site policy.

In addition there is the File Transfer Service FTS as part of the SE. The FTS controls the incoming transfers of an SE. It serves all the VOs and can be compared to the batch queue of a computing farm. The destination SE is always the local SE, the source SE may be inside or outside the site boundaries.

9.1.3 Storage Resource Management Interface

The SRM interface that we adopt is described in great detail in the documents available through the GGF Grid Storage Management Working Group (GSM-WG) webpages [24].

This management API is intended to be used mostly by administrators, the job submission system, and as an internal API between the Grid Services and the SE itself through the POSIX interface.

9.1.4 POSIX-like File I/O

The interaction with the storage element should be transparent to the user through a POSIX-like file I/O API. This interface will not be fully POSIX compliant as we have to relax the POSIX semantics and modify also the POSIX security behaviour. Only a subset will be implemented, as is done in many I/O libraries today.

Figure 9 shows how the I/O works in detail. The I/O client library accepts either LFN or GUID as an input to the API (see section 9.2 for an explanation what the catalogs contain and what GUIDs and LFNs are). The LFN or GUID is presented to the I/O server, which checks with the Catalog Service whether the user is allowed to access the file in the given way, resolving the GUID or LFN into the SURL which is then handed to the local SRM in the process. The actual file handle is acquired through SRM and will be used to access the actual file and serve the data to the client. The detailed internal design is a little more complex than that, including a file cache and also the possibility to read data from a remote SE, routing the request through the remote caches. These details will be given in DJRA1.2, the design deliverable.

There is a lot of room for customisation in this model since all components are pluggable. It is also possible to provide a grid virtual file system such that the system calls can be used for handling files.

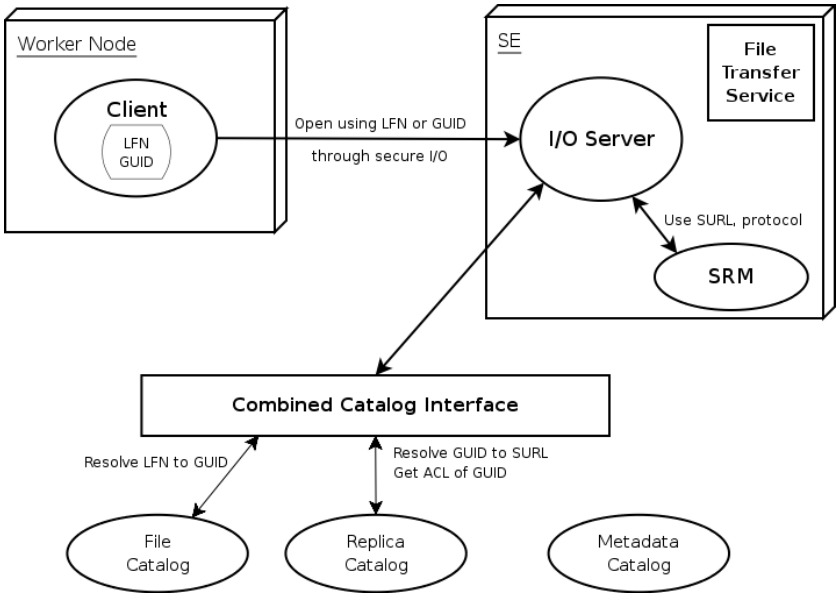

Figure 9: The SE file I/O interactions.

There are existing approaches to provide such a system, such as AlienFS [38]. However as none of them have been deployed in a large scale Grid yet, an evaluation needs to be done first. There is also doubt about whether it is necessary to provide such a virtual file system at all.

The catalog interface that is being called is co-located with the SE on the same site. The Replica Catalog (RC) should have local instances containing information on resolution and authorization (ACL) information on all grid-accessible files in a local SE. Of course this is a deployment choice as the catalog may be on a remote site as well. If the remote site is not accessible, none of the local files can be read. So this situation should be avoided if possible.

File Creation Creating a new file in the SE is preferably done by using the Data Scheduler and File Placement Service (cf. Section 9.3), i.e. an existing file is copied into an SE and registered in the catalog.

If a new file is created through the file I/O open() command, it has to be declared what kind of file the system is supposed to create (see discussion about file access patterns below). A new GUID will be allocated by the system (in order to avoid inconsistencies and the user is not allowed to specify the GUID himself). Associated file metadata has to be added as well. Such a creation is not as efficient as the first method, so it is advised not to use this capability if possible. Some SEs might not provide open for creation at all.

File Access Patterns We differentiate between several different file access patterns:

- **Write once, read many.** These files are read-only, they may not be written, so write methods will not work. They can be replicated safely.
- **Rare append only updates, only one owner.** These files are updateable for writes (appends), and we do not expect concurrency issues, so also such files are replicateable, although detection of conflicts has to be built into the system.
- **Frequently updated, one source.** These files are like parameter files, that may change often, but change only at a single location from where all other replicas are updated. The replicas may be checking whether there is a new version available and pull in the latest version if needed.

- **Frequent updates, many users, many sites.** Such a pattern is too difficult to be managed in general by the grid and is not supported. It is also a rare pattern according to the user requirements so far. Databases provide a much better technology for this usage pattern anyway, so we suggest to store such data in databases, not in files. Even so, multi-master distributed database technology is expensive and error-prone and should be used only if it cannot be avoided.

The I/O methods will exhibit different semantics for each supported pattern. For example the write() method will return an error for write-once read many files.

9.1.5 File Transfer Service

The FTS will take requests and execute them based on the site policies that govern its behaviour. Site administrators can throttle network usage using the FTS policies.

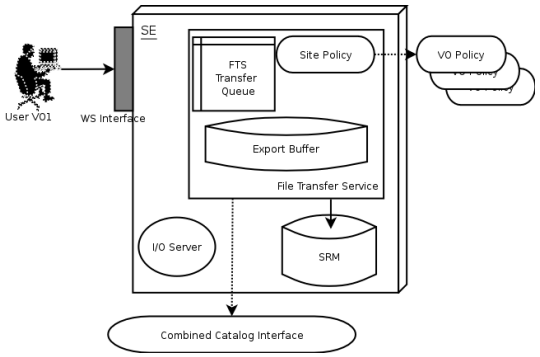

Figure 10: The interactions and components of the File Transfer Service.

The FTS keeps a persistent transfer queue. It works through the transfer queue based on the site policies and manages each file transfer operation. It is responsible for the success of the transfer. The file transfers may be either remote to local or local to local. Usually the FTS receives requests through the FPS, but users and applications might put in transfer requests directly if they are authorized to do so.

Additional policy and optimisation modules may work on the same transfer queue, updating the order of the requests to be processed based on site policies.

The FTS calls out to the Replica Catalog's ACL information in order to make sure that a user has actually access to the source data and is allowed to write to the destination. If this check fails the transfer will be refused.

The FTS can also perform protocol translation. If two SEs cannot communicate directly because they do not support any common protocol, the FTS streams the data from the source through an in-memory process using a source protocol and streams it out again to the target using a target protocol. If the protocols do not support streaming, the file may be cached locally if space is available. Only if all these options are exhausted does the system give up. In principle, each SE should support at least GridFTP as a common protocol, and this should only be necessary if for some reason GridFTP transfers do not work and a fall-back protocol is used.

9.1.6 Direct SE Access

The SE also needs to allow for direct file upload and download through some protocol. This also needs to be properly secured through the catalogs. Such access also needs to be authorized first.

If these transfers go through the File Transfer Service FTS, then the FTS will take care of choosing the most appropriate protocol and mapping the user as necessary.

If a direct access is needed for other reasons through a specific protocol, there needs to be a mapping between the user contacting the SE through the server exposing the given protocol (like GridFTP) and the internal I/O server user serving the file. For this case the user gets a temporary export cache area owned by the user's locally mapped userID, where the files requested will be cached by the I/O server if authorization is successful. This cache is then accessible by the service that provides the requested protocol, accepting the user's credentials. Such a local cache is maintained by the FTS anyway so this is simply a specialised use of the FTS component. There are security implications though (see below for a discussion).

9.1.7 SECURITY

Many mass storage systems do not provide native support for ACLs. This is the reason to keep them (GUID-based) in an external catalog, the Replica Catalog. It acts also as an authorization service for files in this context. In terms of security, the work-flow between the SE and the authorization information contained in the RC is simple: internally the SE services all run as the same grid master user, being fully authorized themselves to access the data stored in the SRM. However, before actually performing an operation on behalf of the user, the RC is consulted to see whether the user is actually authorized to perform the given operation.

The client will contact the SE interfaces using his credentials, delegating the rights to the server to contact the Combined Catalog Services (see detailed description in section 9.2.3). To the user this operation is completely opaque, the SE client API is as if the SE itself would contain all the necessary catalogs itself and simply would have the ability to deal with ACLs. This way SRMs not having ACL capabilities can also be used as fully secured systems. As mentioned above, usually all files are owned by the I/O server. If the user is authorized to perform a given operation, the server will extract the data from the SRM and serve it to the client.

However, we foresee the possibility to map users to another owner as well if they happen to have data in the local store as another user. The mapping can be either explicit through the user's DN or through a role or capability given in the user's VOMS credential. The disadvantage of using the DN only as the mapping mechanism is clearly that if the user intends to share files with others through ACLs, this will not be possible on this particular storage because nobody else is being mapped to this user. Also the user would be unable to read anybody elses files. So the detour through VOMS groups or roles is preferred: if someone wants to have their existing data exposed through the grid which is stored as some specific user, this user is mapped into a VOMS group, which is directly managed by the user, giving group membership to others if data is to be shared. The user can specify to be in the group or not as well - depending whether data inside his or her own space or not should be accessed. The clear disadvantage of this approach is that if the user needs data from both her own space and from the VO space in the same session, this will not be possible if access rights on the local mass store are incompatible.

9.2 FILE AND REPLICA CATALOGS

In the Grid the user identifies files by logical file names (LFNs). The LFN is the key by which the users locate the actual locations of their files. We refer to file *replicas* (as opposed to file copies) if the instances of a given file are being tracked by the Replica Catalog (RC).

The replicas are identified by Site URLs (SURLs). Each replica has its own SURL, specifying implicitly which Storage Element needs to be contacted to extract the data. The SURL is a valid URL that can be used as an argument in an SRM interface (see section 9.1). Usually, users are not directly exposed

to SURLs, but only to the logical namespace defined by LFNs. The Grid Catalogs provide mappings needed for the services to actually locate the files. To the user the illusion of a single file system is given.

To maintain this illusion, the Grid data management middleware has to keep track of SURL - LFN mappings in a scalable manner. Also, the identifier of a file entity has to be kept unique at all times. In order to achieve this, a Global Unique Identifier (GUID) is given to each file when it is created on the Grid.

9.2.1 FILE NAMES

The logical namespace needs to be unique. We have to agree on some properties of the logical file names, as it is the case also for distributed file systems. If we take AFS as an example, it declares its root to be /afs followed by the AFS cell name [26]. Similarly, we suggest to apply the same idea to the Grid file namespace, i.e. start with `/grid` followed by the virtual organisation name. Just like for AFS cells, the VO name has to be unique as well. We suggest a domain name or host name which is actually a valid entry in the DNS. Under this namespace each VO can define it's own structure to prevent conflicts.

We have the following file names in the Grid:

LFN Logical File Name: A logical (human readable) identifier for a file. LFNs are unique but mutable, i.e. they can be changed by the user. The namespace of the LFNs is a global hierarchical namespace, as explained above. Each Virtual Organisation has its own namespace.

GUID Global Unique Identifier: A logical identifier, which guarantees its uniqueness by construction (based on the UUID mechanism [28]). Each LFN also has a GUID (1:1 relationship). GUIDs are immutable, i.e. they cannot be changed by the user. Once a file acquires a GUID it must not be changed otherwise consistency cannot be assured. GUIDs are being used by Grid applications as immutable pointers between files. If these should change, the application may suddenly point to a wrong file.

SURL The Site URL specifies a physical instance of a file replica. Also referred to as the Physical File Name (PFN). SURLs are accepted by the Storage Element's SRM interface. Their schema is therefore `srm`. A file may have many replicas, hence the mapping between GUIDs and SURLs is one-to-many. (The Storage URL StURL is another term used by the SRM specification, for the actual file name inside the storage system. To the Storage, the Site URL is a logical name and the StURL is the real location of the file on disk.)

So the LFN and GUID are both unique but with the important difference that the LFN is human-readable and mutable, whereas the GUID is neither. Some VOs may enforce the policy of immutable LFNs in which case there is no semantic difference, except that the logical namespace of the LFNs provides a means of hierarchically grouping the files whereas the GUID namespace is flat.

The LFN is the name of a file in the VO-internal logical namespace, organised in a hierarchical directory structure. A valid LFN might be a string like

```
/grid/myVO.org/production/run/07/123456/calibration/cal/cal-table100
```

The LFN must be unique within a VO and should be accessible to the entire VO. The File Catalog allows symbolic links to be set to LFNs, their semantics being analogous to those in the Unix file system. Similarly to symbolic links in Unix, LFN may have a nonzero number of symbolic links. Symbolic links have to be specified using absolute paths. They have the same semantics as the Unix filesystem symbolic links, i.e. their consistency is weak and they might point to non-existing LFNs. Any operation on the target LFN does nothing to update the link; symbolic links can create cycles, etc.

The internal GUID does not need to be exposed to the users, who will usually only see the human readable LFN. However, a file may have no LFN at all, only a GUID. The introduction of the GUID allows us also to distribute the catalog across the wide area and to catch accidental duplicate LFNs should they occur. An example case where this might occur is when a batch farm producing some output is disconnected from the wide area network and registers a new file (and a new LFN) in its local File Catalog. Upon reconnection, the local File Catalog instance tries to re-sync with the rest of the world, and finds the LFN already registered. The clash can be dealt with by some configurable policy, the easiest of which would be to mail an administrator. The GUID then gives a guaranteed-unique handle that the administrator can use to reference the file while dealing with the clash. The typical resolution would be to assign the file a different LFN. In general, the application should take reasonable steps to ensure that the LFN is unique; the process above is only for recovery purposes. In analogy to Unix file systems one might say that the GUID is like the inode.

The SURL is what is used to access the file at a Storage Element (SE). Usually, users should not need to be aware of the SURLs they are using, only the logical filename and its directory structure are of relevance to them.

9.2.2 Other Services Used

The Metadata Catalog is used by the Combined Catalog interface to implement virtual directories. The Combined Catalog interfaces to all catalogs. The authentication and monitoring services may be used as auxiliaries.

9.2.3 Catalog Services

The File Catalog (FC) and Replica Catalog (RC) both make part of the GUID – LFN – SURL mappings accessible (see Figure 11). The File Catalog interface exposes operations only on the mappings that it manages, as does the Replica Catalog interface. The Replica Catalog interface gives access to the GUID – SURL mappings identifying all replicas of a given GUID. All the rest of the mappings are kept in the FC. Operations that involve both catalogs and potentially also the Metadata Catalog are exposed through a Combined Catalog interface. The reason for this interface decomposition is to have service interfaces

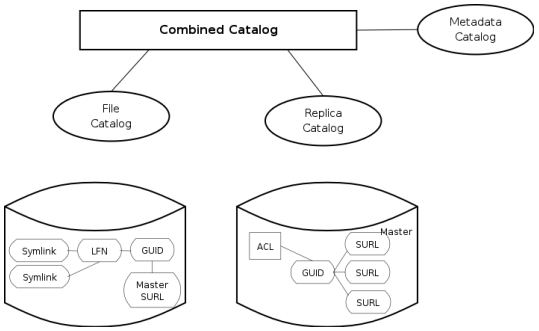

Figure 11: Catalog Services with the accessible mappings.

with well-defined semantics which may be implemented by many parties. In this model, a possible scenario is that all three interfaces may be implemented over a single central database. Another possibility is that the File Catalog is central whereas the Replica Catalog is distributed over many sites and synchronised through some database replication scheme. Also, either the File Catalog, Replica Catalog or most commonly the Metadata Catalog might be implemented and controlled by the VO directly. If the implementation provides the proper interfaces, it can participate in the Data Management System and

need not provide all the other functionality as well. Actual deployment also depends on the needs of the particular VO. It is possible to share services across VOs, but we expect that each VO has its own set of catalog services.

The need for all these possibilities in to choose between different implementation and deployment models and was one of the reasons to go for a service oriented architecture in the first place (see Section 4). In the following we discuss some details of the catalog interfaces and semantics.

9.2.4 OPERATIONS

There are three interface categories, each exposing a well-defined set of operations.

File Catalog The File Catalog operations deal with operations on the logical namespace such as making directories, renaming files and symbolic links.

Replica Catalog List, add and remove replicas based on a GUID.

Combined Catalog All operations that need synchronised access to at least two catalogs (including the Metadata Catalog) such as creation of virtual directories, creation of new files (including the master copy and GUID) and deletion.

All catalog interfaces expose bulk operations as part of the interface. They increase performance and optimise interaction with the Grid services.

The Combined Catalog needs to maintain a persistent state of all operations it performs across catalogs in order to make sure that the operations only occur in a synchronised manner.

9.2.5 SCALABILITY AND CONSISTENCY

The File Catalogs that have been deployed to date are all deployed centrally and therefore are a single point of failure. The central catalog model has obviously excellent consistency properties (concurrent writes are always managed at the same place) but it does not scale to many dozens of sites. There are two possibilities to solve this issue:

- Database Replication. The underlying database is replicated using native database replication techniques. This may mean a lock-in to a vendor-specific solution. Currently commercial database vendors like Oracle provide multi-master database replication options which may be exploited for such a purpose.
- Lazy database synchronisation exploiting the specific semantics of the catalogs using messaging to propagate the updates. Reliable messaging technologies are available commercially (just like replication for database technologies) and there are some solutions also in the open source domain. The File and Replica Catalog semantics are rather simple and very specific for catalog write operations, so that every time a local write operation occurs, it can be distributed through a message queue to all known and available remote catalogs.

The EGEE middleware will be able to accommodate both solutions. Consistency might be broken in both models i.e. it is possible to register the same LFN in two remote catalogs at the same time such that a conflict will occur. The reconciliation techniques apply in both cases. In the second case we can be specific to the semantics of the system and exploit the uniqueness of the GUIDs.

9.2.6 THE MASTER REPLICA

Currently we do not expect the files to be updated. In a distributed system to keep track of all replicas in a consistent manner is a nontrivial task that the middleware should not need to deal with from the start. However, a placeholder is needed to enable such functionality. The master replica as present in the File Catalog, is the only replica where update operations are allowed. This is also the master source for replications. If the master replica is lost, it might be recovered from other replicas or not, based on policies. A master replica should always be kept on a reliable, "strategic" SE.

Of course the master replica is also kept in the Replica Catalog and is flagged as such. (This is needed since not all files might have an LFN). The double bookkeeping helps with the resilience of the services: Should the Replica Catalog be out of order, the File Catalog can still offer at least one safe location of a given file.

9.2.7 DIRECTORIES

We define the concept of a directory in the LFN namespace. The directory may be manipulated just as in a normal file system. It can be listed in the same manner, new entries may be made to it or existing ones renamed and removed. The LFN contains the directory structure that can be navigated in such a manner to reference all the logical files for a given VO. The HEPCAL DataSet concept maps well to a directory in the logical namespace.

In order for a new LFN to be created, its parent directory needs to exist first. A new directory may be created through a simple API call (`mkdir` as usual). A user can copy logical files from other logical directories into their own directory. Symbolic links may be set to other directories.

9.2.8 VIRTUAL DIRECTORIES

A virtual directory is a special directory created from a metadata query. A user defines a metadata query whose output is a set of logical files that are then made accessible through the virtual directory as symbolic links. The virtual directory contains an enumerated list of files, all of which are symbolic links to the files returned by the user's query. Only a limited set of operations are available in such directories. The result of a query is stored in the File Catalog, so whenever the user issues the list command, the same stored results are retrieved. In order to refresh the contents of the directory, a virtual directory refresh has to be issued explicitly. The advantage is that the user has to explicitly invoke remote catalog calls, they are not invoked "accidentally". Also, since the actual data may change, the user has the possibility to see how a query evolves over time.

9.2.9 DATASETS

The functionality of having a dataset as described in the high energy physics requirement document [9] is not explicitly supported by this set of services since the catalogs described here are purely file-based. However, for applications having files as the data granularity most of the functionality is available by using the logical namespace mechanisms for directory and virtual directory, covering a large fraction of the Use Cases.

- A directory is an explicit dataset by inclusion. Files may be added, removed by the usual mechanisms.
- Datasets by reference may be a directory containing symbolic links to other files.
- A directory may have "real" files as well as symbolic links.

- A virtual directory is a dataset which is created from a metadata query.

Datasets that are more complex, containing object references and metadata as well as references to files are usually application specific and we expect them to be managed by a higher-level service or through the metadata catalog. VOs may provide a specialised metadata service to manage complex datasets and nontrivial operations on datasets. Such a catalog may make use of the EGEE catalog interfaces internally, hiding the complexity from the user.

9.2.10 SECURITY

By imposing ACLs on the filesystem the security semantics are straightforward. This should also help in avoiding concurrency issues when writing into the catalog since each user will have only limited access rights in the LFN namespace and there should be only a finite set of administrators per VO who have full access rights for all of their LFN namespace. The probability of two users with the same access rights to write into the catalog in the same directory in a distributed system is therefore low.

The Replica Catalog interface exposes the operations on the file ACLs. There are two possibilities of how ACLs may be implemented.

- **POSIX-like ACL** The POSIX semantics follow the Unix filesystem semantics. In order to check whether the user is eligible to perform the requested operation, all of the parent ACLs need also be parsed.
- **NTFS-like ACL** The Microsoft Windows semantics are simpler, i.e. the ACLs are stored with the file and the branch has no effect on the ACL. These are “leaf” ACLs, only operating on the file itself.

In a distributed environment, the NTFS-like semantics are simpler to track and are probably more efficient. The Replica Catalog interface exposes all operations that deal with querying and setting of the file ACLs. It acts as the authorization authority for file access and is called upon by other services such as the File Placement Service to enforce ACL security.

9.3 DATA MOVEMENT

In this section the architecture of the Data Management System is described, defining it’s responsibilities and the interactions between its services. The data management services provide scalable and robust managed data movement. Files are scheduled to be moved between sites reliably. Scalability and manageability have driven the service decomposition described in this section. It is important to distinguish services owned and managed by the VO from those owned and managed by the site administrators.

The services making up the Data Management System are: The Data Scheduler (DS), the Transfer Fetcher and the File Placement Service (FPS). Another component described in this section is the File Transfer Library (FTL). Additional higher-level services making use of these basic services are discussed at the end of the section.

The Data Scheduler is a top-level service, keeping track of data movement requests in a VO that are being submitted directly by the user through a portal or user interface or by computational jobs submitted to the Workload Management Service. The Transfer Fetcher polls the Data Scheduler and fetches transfers whose destination is the local site for the given VO, inserting new requests into the File Placement Service. The File Placement Service coordinates the transfer performed by the File Transfer Service and makes sure that the File and Replica catalogs are updated properly.

This service breakdown has been chosen to keep the services modular and dedicated to a well-defined task. The File Transfer Service FTS is the lowest-level service used here (see section 9.1.5). It deals only

with the transfer of files between Storage Elements and between Worker Nodes and Storage Elements. The destination SE is intrinsically given by the FTS which is part of the given SE. Each SE runs only one FTS. The File Placement Service FPS has the task to coordinate the transfer with proper registration in the Catalogs. It monitors the progress of a transfer in the local FTS and will make sure that the catalogs are properly updated, retrying the operation according to well-defined VO policies if necessary. Each VO has an FPS per site. The user submits requests for transfer (implicitly including the proper catalog operations) through the File Transfer Library API to the local FPS. If the transfer request does not have a local SE as it's destination, the FPS forwards the request into the VO's global Data Scheduler. So it keeps it's task well-defined and simple.

The Transfer Fetcher is a component running at each site, periodically polling the DS whether there are any pending transfer requests that have the local site as it's destination. If yes, these requests are retrieved from the DS and forwarded to the local FPS. Once the FPS completes the request, the fetcher will update the DS to signal the completion of the request.

The DS can also be contacted directly by the user through the user interface. It keeps a persistent queue of all requests. Other processes such as policy managers and optimisers may operate on the same queue to refine the requests.

This service breakdown, which is modelled after well-understood mechanisms already in use by schedulers, ensures that the services are kept simple enough to maintain a high reliability of the overall system. All of the services sketched out above are described in detail in the following sections.

9.3.1 Other Services Used

The other data services are all used, i.e. the Storage Element and Catalog services, as well as services related to security, monitoring and accounting as helper services.

Services interacting with the Data Management subsystem are the user through the Grid Access Service or directly through a job, and the Workload Management System.

9.3.2 Data Movement Services

This section describes the architectural design, responsibilities and interactions of the components in the Data Management subsystem.

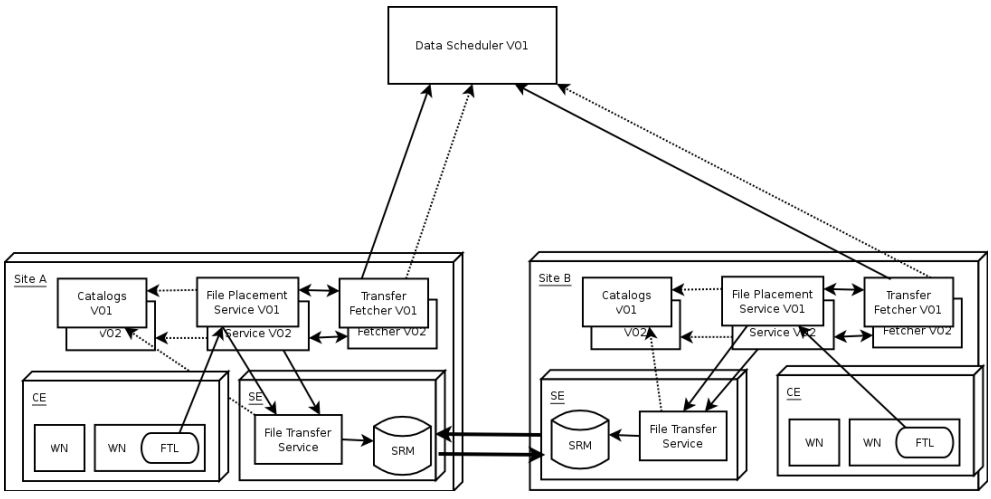

Figure 12: Architecture overview of the Data Management Services. The dotted arrows represent queries, the solid arrows represent requests.

The overview of the services is shown in Figure 12. The external services shown in the figure are Catalog interfaces (see section 9.2), the Storage Element (SE), the Computing Element (CE) with the Worker Nodes (WN). Some services, such as the CE, SE, FTS, may be tied to a site serving all VOs. Others are deployed such that each Virtual Organisation (VO) runs their own service (Data Scheduler, File Placement Service, Catalogs).

The Data Management subsystem has the following components:

Data Scheduler From the VO's point of view the Data Scheduler is a single central service. It may actually be distributed and there may be several of them, but that depends on the implementation. It is responsible for scheduling and keeping track of the data transfers and catalog operations between multiple sites.

Transfer Fetcher The Fetcher polls the available DS for a given VO and checks for transfers where the target is an SE at a local site. It takes the transfers from the DS and submits them into the FPS. (Pulling requests from DS, pushing into FPS). So it acts as the connecting service between the global DS and the local FPS.

File Placement Service FPS There is an FPS running at each site making one logical instance per VO. It coordinates the file transfers and the catalog operations, exposing atomic file replication operations to it's clients.

File Transfer Library FTL The FTL provides the API interface to the data management operations available to Grid clients. It manages the client's communication with the FPS and the FTS if any transfer operations are to be made between the client's Worker Node and a local SE.

In the following, we describe each service in more detail and explain their control flow, look at their components in more details and discuss actual use cases.

9.3.3 DATA SCHEDULER

The Data Scheduler (DS) is the high-level service that keeps track of most ongoing WAN transfers inside a VO. In principle, the DS schedules data movement based on user requests. These requests are restricted to a well-defined set with well-defined semantics. This restriction is to guarantee a working system at the outset. At a later stage, the DS might be extended to accept more dynamic kinds of requests by virtue of a data job description language, enabling it to do data job scheduling coordinated together with the WMS.

Users and the job scheduling system will contact the DS in order to move data in a scalable, coordinated and controlled fashion between two SEs. The DS has four components (per VO) as shown in Figure 13:

VO Policies Each VO can apply policies with respect to data scheduling. These may include priority information, preferred sites, recovery modes and security considerations. There may be also be a global policy which the VO, by its policy, may choose to apply. Global policy is usually fetched from some configurable place. The same global policy may apply for many VOs.

Data Scheduling WS Interface The actual service interface that the clients connect to, exposing the service logic. All operations that the clients can use are exposed through this service.

Task Queue The queue holding the list of transfers to be done. The queue is persistent and holds the complete state of the DS, so that operations can be recovered if the DS is shut down and restarted. This queue is the heart of the DS, all services and service processes operate on this persistent queue.

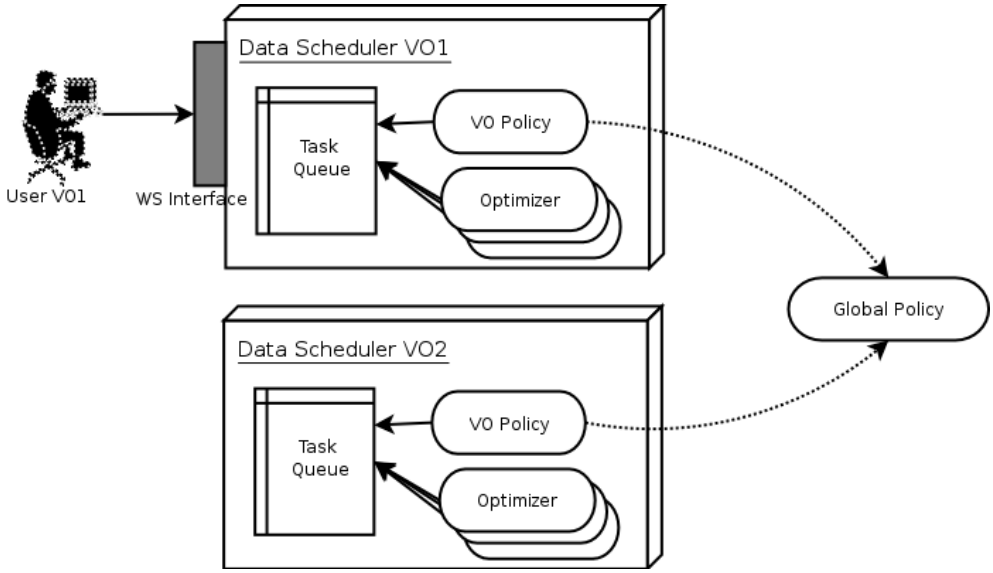

Figure 13: The Data Scheduler

Optimisers A set of modules to fill in missing information (like the source to copy from), or modifying the details of a request based on some algorithm (such as the protocol or the source replica to be used).

The DS is pictured in Figure 13 logically as a single instance. However, there may be several DS's for the same VO deployed in the Grid. The DS is contacted by the File Placement Service from the Grid sites (see Figure 12).

9.3.4 TRANSFER FETCHER

The Fetcher periodically polls the Data Scheduler to see whether there is any work to be done and submits matches to the FPS. It monitors the progress of the FPS tasks that it has submitted and informs the Grid Monitors and also the DS of the progress of these tasks. Once a task has finished, the Fetcher will notify the DS. This is a very simple lightweight service that runs a logical instance per VO at each site. The actual implementation may be such that a single service serves many VOs, allowing for a flexible deployment model (one per VO or many VOs per service).

The Fetcher also may also apply site policies and local VO policies.

9.3.5 FILE PLACEMENT SERVICE

The FPS receives requests from the user through the File Transfer Library API (see below) or through the Transfer Fetcher. If the transfer request has not a local SE as it's destination, the request is forwarded to the Data Scheduler (via the local Transfer Fetcher).

Once a request is accepted, the FPS synchronises operations on the File and Replica Catalog and the File Transfer Service so that the information in the catalogs is correct. It makes sure that a transfer performed by the FTS is also properly registered into the catalogs. If the catalogs or the FTS cannot be reached, the request is re-tried based on a user and site-definable policy (number of retries, timeouts). The FPS keeps track of all unfinished operations persistently in case it has to be restarted itself.

VOs may apply policies to data transfer at this level. These policies may include prioritisation, throttling, etc. Site policies may be applied at the FTS level.

9.3.6 File Transfer Library

The FTL exposes the client API the applications can use in order to communicate with the Data Management System.

The FTL is also the users' interface to access grid files directly through a POSIX-like I/O. In addition it exposes an API to copy local files into a Storage Element or extract files onto a local path. When a client (user or application) performs a file transfer to or from the SE to the local disk, it contacts the local File Placement Service from the worker node. The File Transfer Service will eventually schedule the transfer and contact the client's FTL to coordinate the transfer. The purpose of this logic is to allow the File Transfer Service to treat the transfer in a similar way to the one between two Storage Elements, and for the user to be able to control the transfer from the outside if necessary. Failures may be logged at the service level, and for retries and policies the same mechanisms apply as for inter-SE transfers.

9.3.7 Additional Higher Level Services

Having the basic set of data management services in place, it is straightforward to put additional convenience services in place, usually in the application layer. An example is a component that automatically schedules data transfers based on some trigger or event. The event may be application specific or it may be linked through the catalogs (new entries) or from the SE (new data) or other monitoring services. The auto-scheduling service would place new requests into either the local FPS or the global DS periodically whenever the event is triggered. It may use the FTL interface to achieve the scheduling task and can focus on implementing its triggering mechanism.

Additional application-domain services may be constructed on top of the data management services in a similar fashion.

9.3.8 Security

The FPS and Transfer Fetcher services manage all transfers and catalog interactions. In order to be secure, the system has to check whether the requester is allowed to read the data from the source SE and is allowed to write the data to the destination SE. The component that responds to authorization inquiries concerning file access in the logical namespace (also coupled to the GUIDs) is the Replica Catalog.

The detailed security design will be finalised in our design document deliverable.

9.4 File Metadata Catalog

All metadata is application specific and therefore all metadata catalogs should optimally be provided by the applications and not the core middleware layer. There can be callouts to these catalogs from within the middleware stack through some well-defined interfaces, which the application metadata catalogs can choose to implement.

In this context, we consider metadata only in the context of file-based metadata, i.e. metadata that is related to the files stored in the Grid, with either the LFN or the GUID as the key binding the metadata and the File and Replica Catalogs together.

File-based Metadata Catalogs contain application-specific information (arbitrary and extensible set of attributes) about the contents of the available files. These metadata may be used to query the Metadata Catalog when searching for datasets to be processed on the Grid.

For any arbitrary existing metadata catalog that contains file-based information it should be possible to interface it to the Grid. A necessary ingredient for such a mechanism is a standard interface for interacting with a metadata catalog, which then has to be implemented on top of each existing catalog. The metadata catalog interface for file-based metadata can be specified to offer a well defined set of operations. File-based metadata always assigns metadata to logical file names or GUIDs, and has therefore more specific semantics than generic application metadata.

9.4.1 Other Services Used

The File Catalogs make use of the Metadata Catalog. The Metadata Catalog may have callouts to the security services (authentication and authorization).

9.4.2 Services

Generic metadata service interface can be offered through a generic Grid Database service that makes data accessible through the Grid. Such services have been tested in previous projects, like project Spitfire as part of the EU DataGrid project and the OGSA-DAI project of UK e-Science.

The specific service for file-based metadata defines an interface which may perform two simple operations: Queries returning a list of LFNs or GUIDs and inserts based on LFNs and GUIDs as keys.

The specific interface which returns a set of LFNs (limited to a maximal size) for a query takes a string as its argument so the schema and content of the Metadata Catalog needs to be known by the user issuing the query. It cannot be specified by the Grid middleware. Here we only specify the generic metadata interface and its possible failure modes. This interface is then used by the File Catalog and is implicitly exposed through the metadata operations of the Combined Catalog interface, which also involves virtual directories.

10 Use Cases

This section describes the interactions between the individual Grid services discussed above required to execute a goal of the end user. These interactions are expressed as sequence diagrams following the Unified Modelling Language (UML) [46].

Based on the application requirements presented in Section 3 three generic, simplified use cases that allow an easily comprehensible description of the high-level Grid services have been chosen:

1. **Grid Login**: acquiring access to the Grid, in particular the services required to perform work on the Grid
2. **Generic Production Job**: a job producing and storing data on Grid storage is submitted
3. **Generic Analysis Job**: the data produced by a previous production job is analysed (accessed) by a Grid job.

For these Use Cases is is assumed that the user has already obtained appropriate Grid certificates from a CA and joined a VO which allows him/her to perform these tasks.

Note, that the Use Cases separately depict the policy assertion to better illustrate this important issue. As discussed in Section 5.2, policy enforcement will in most cases not be implemented as a separate service but be part of the service for performance reasons.

10.1 GRID LOGIN

Figure 14 shows the typical sequence of actions when a scientist logs in to the Grid and starts a working session.

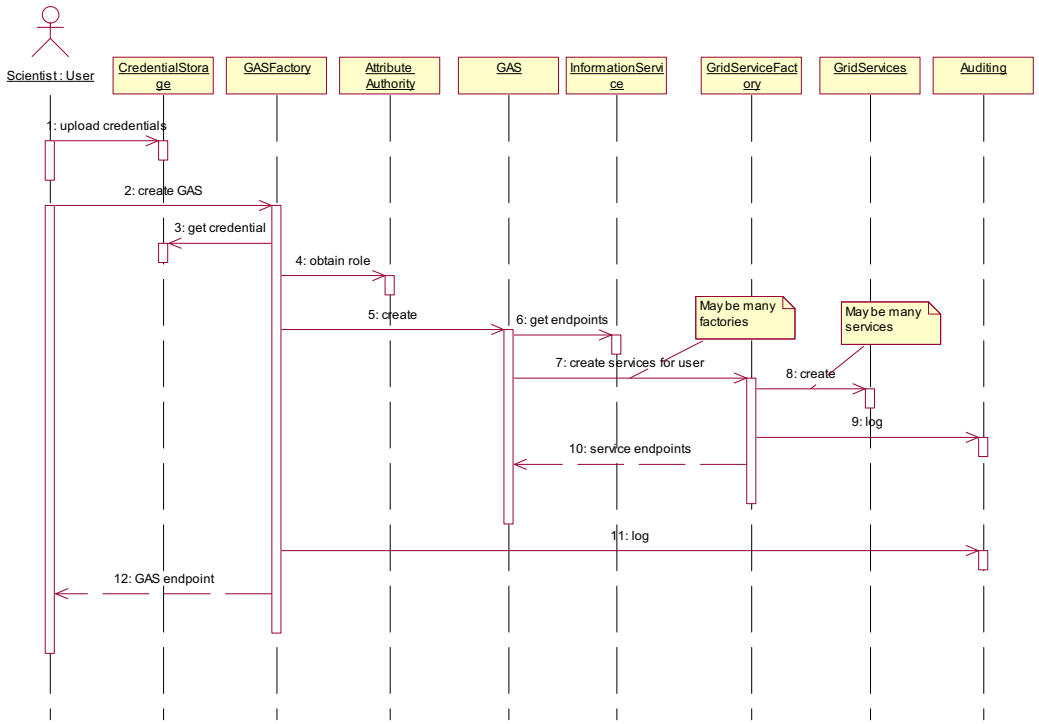

Figure 14: GridLogin

1. The user stores his/her credentials in a credential storage.
2. The user contacts the GAS factory (whose endpoint is known via some bootstrapping mechanism such as a configuration file, webpage, etc.) to create a new instance of the GAS which will provide the environment for the user.
3. The GAS factory retrieves the user's credential from the credential storage.
4. The GAS factory contacts the attribute authority to obtain the role and privileges of the user.
5. The GAS factory instantiates a new GAS instance, passing on the user credentials.
6. Via the information service, the GAS retrieves the endpoints of the service factories available to the user (based on his/her privileges).
7. The GAS contacts the appropriate Grid service factories which in turn
8. create the appropriate Grid service instances which will be available to the user.
9. These actions are logged in the auditing service.

10. The GAS stores all the endpoints of the created services and is ready to be contacted by the user.
11. The GAS factory logs its activity in the auditing service and
12. returns the GAS endpoint to the user.

From now on, the user may interact with the GAS which will invoke the appropriate services to achieve the necessary tasks.

Note, that it is not compulsory to use the GAS; the user may contact and manage the individual Grid services on his own. Also, if a GAS already exists for the user in his/her role, this GAS may be reused, reducing the Use Case to having the GAS factory simply returning the appropriate GAS endpoint immediately.

10.2 GENERIC PRODUCTION JOB

Figure 15 shows the typical (simplified) sequence of actions when a scientist submits a job that produces data stored on Grid storage.

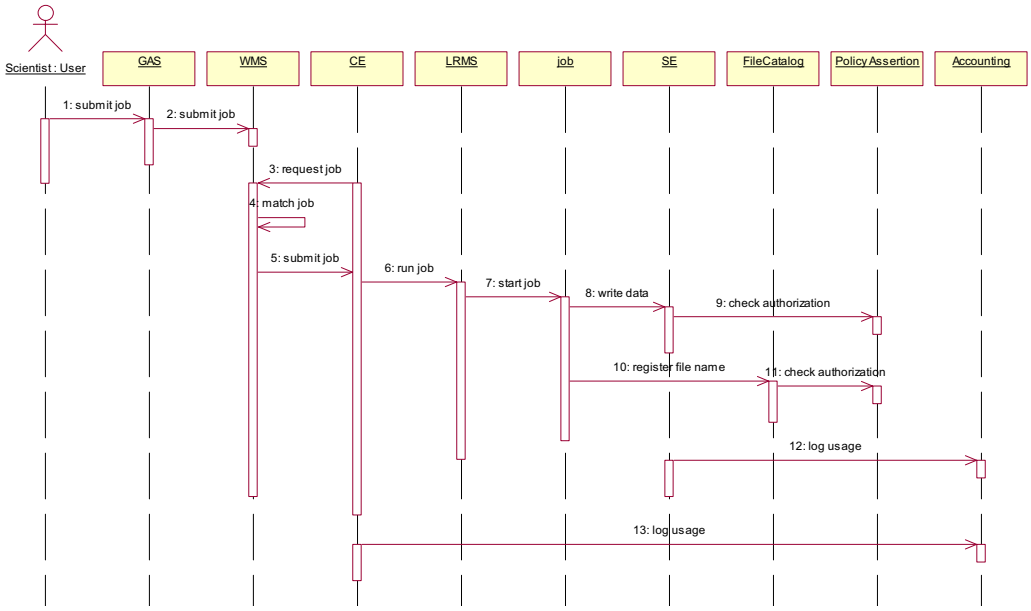

Figure 15: Production Job

1. The user contacts the GAS to submit the job.
2. The GAS submits the job to the appropriate workload management system.
3. A CE requests a job from the WMS.

4. The WMS performs a matchmaking among the available jobs and CEs requesting jobs.
5. The job is submitted to a suitable CE.
6. The CE submits the job to a LRMS
7. Eventually, the job runs on a worker node, guarded by a job-wrapper.
8. The job is producing data which is stored on a Storage Element.
9. The SE checks whether the user is authorized to write data.
10. After the data has been produced, the file is registered in the file catalog, which
11. checks whether the user is authorized to register the file.
12. Accounting information is logged from the SE.
13. Accounting information is logged from the CE.

All interactions are also logged in the auditing service, which is omitted here to reduce the complexity of the diagram.

10.3 GENERIC ANALYSIS JOB

Figure 16 shows the typical (simplified) sequence of actions when a scientist submits a job that needs to analyse data stored on Grid storage.

1. The user contacts the GAS to submit the job; the job description contains the logical name of the data to be analysed.
2. The GAS submits the job to the appropriate workload management system.
3. The WMS contacts the file catalog to retrieve information on the location of the file.
4. The file catalog checks whether the user is authorized to get information on the file.
5. A CE requests a job from the WMS.
6. The WMS performs a matchmaking among the available jobs and CEs requesting jobs taking into account the location(s) of the requested file.
7. The job is submitted to a suitable CE; the job also carries information on how to access the requested file.
8. The CE submits the job to a LRMS
9. Eventually, the job runs on a worker node, guarded by a job-wrapper.
10. The job reads the data which is stored on a Storage Element.
11. The SE checks whether the user is authorized to read the data.
12. The results of the analysis are stored on a SE.
13. The SE checks whether the user is authorized to write data.
14. After the data has been produced, the file is registered in the file catalog, which

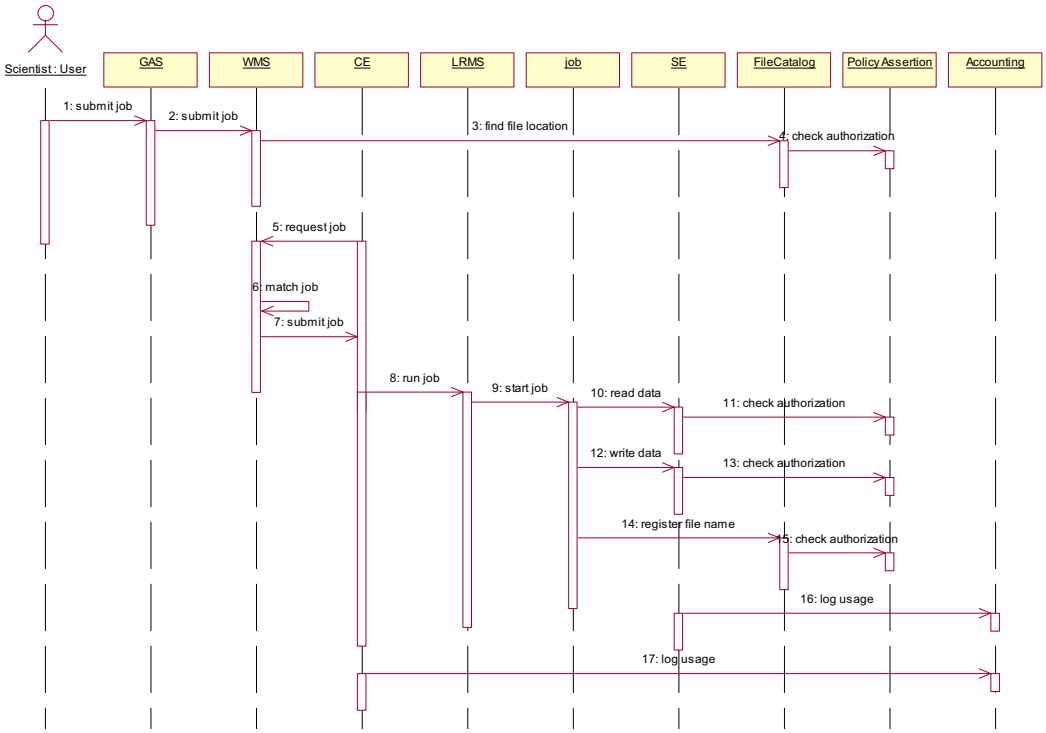

Figure 16: Analysis Job

15. checks whether the user is authorized to register the file.
16. Accounting information is logged from the SE.
17. Accounting information is logged from the CE.

All interactions are also logged in the auditing service, which is omitted here to reduce the complexity of the diagram.

11 ISSUES

11.1 SITE PROXY

As discussed in Section 5.2.2, our design must recognise and respect policies defined by the local sites. Such policies may affect connectivity as resources (storage, worker nodes) may be on a private network, NAT devices or firewalls may block incoming and/or outgoing connections and certain port intervals may be blocked. [31].

In order to tackle the problems that arise when connectivity is restricted by the resource owner, provision for a *Site Proxy* component was identified early in the design process.

First, it was envisaged that a site proxy would act as a virtual endpoint for service-level translation device/routing of messages between the Internet and the actual endpoint on a local/private/shielded network.

However, such as service would effectively override/bypass local policy, which is not in line with the overall security design philosophy. Furthermore, the middleware services communicate over standard protocols for which proven solutions already exist.

Therefore, our initial approach is not to build a site proxy ourselves. Instead, we will ensure that our service container(s) and services themselves can be configured such that the local resource owner can control how the services communicate:

- Only using ephemeral (dynamically allocated) ports in a certain port range
- Advertise endpoint URLs with hostname/port numbers different than the physical local addresses and logical hostname
- Route all SOAP over HTTP(S) traffic via a SOCKS server, provided by the resource owner.

For the longer term, we will investigate alternative solutions to this problem. For instance, other transport protocols such as BEEP [35] may be used. One could also envision a site proxy as a front-end service to the local network access policy, with the capability to authorize and trace network access by modifying that policy dynamically. For example, before initiating a data transfer, the application would need to request for the creation of a network path on which it can send or receive. The site proxy service would then alter the firewall configurations to allow traffic through, and log the event for auditing purposes.

11.2 SERVICE COORDINATION

In service-oriented architectures (see Section 4), services expose a well-defined set of operations to the clients where the operations are usually atomic. Atomicity means that an operation either succeeds or fails completely; there are no partial failures or partial successes. As an example, consider the File Placement Service described in Section 9.3.5. It has to coordinate the File Transfer Service and Catalog Service operations in order to expose an atomic behaviour.

So in service-oriented architectures we often build new services by aggregating other services. In a distributed environment, however, there is no guarantee that a service may be contacted at all times. This means that the service which is coordinating operations of other services has to have built-in policies on how to deal with failures.

The policies depend on the semantics of the service. In our example of the FPS, it makes sense to re-try to contact the File Transfer and Catalog Services at certain intervals (until a timeout is reached) to try to complete the operation. However, if we try to copy a file first and fail with the catalog operation, should the copy be removed in the process? (Our answer is yes.)

There are several proposed specifications to deal with the issue of service coordination [52]:

- **WS-Coordination** describes an extensible framework for providing protocols that coordinate the actions of distributed applications or services.
- **WS-Transactions** describes coordination types that are used with the extensible coordination framework described in WS-Coordination. It defines two coordination types: WS-AtomicTransactions and WS-BusinessActivity. Either of these can be used when building applications requiring consistent agreement on the outcome of distributed activities.
- **WS-ReliableMessaging** describes a protocol that allows messages to be delivered reliably between distributed applications in the presence of failures.

It is an open question how these specifications can be used to achieve the desired functionality of our services. Such an evaluation will have to be made on a case-by-case basis.

11.3 STANDARDS

11.3.1 WSRF RESOURCES

Very recently, the Grid community has proposed the notion of Resources in the Web Service context by proposing a Web Service Resource Framework (WSRF) as a standard framework for building a rich set of services managing stateful resources [13]. The Global Grid Forum (GGF) bases its Open Grid Service Architecture (OGSA) on the WSRF model [16].

The motivation to introduce WSRF is as follows [47]: Web services must often provide their users with the ability to access and manipulate state, i.e., data values that persist across, and evolve as a result of, Web Service interactions. And while Web Services successfully implement applications that manage state today, we need to define conventions for managing state so that applications discover, inspect, and interact with stateful resources in standard and interoperable ways. The WS-Resource Framework defines these conventions and does so within the context of established Web Services standards.

The WS-Resource Framework (WSRF) is a set of six Web Services specifications that define what is termed the WS-Resource approach to modelling and managing state in a Web Services context. Not all of these specifications have been made public yet.

The WS-Resource approach has been introduced in order to declare and implement the association between a Web Service and one or more named, typed state components [51]. In this approach, state is modelled as stateful resources and codify the relationship between Web services and stateful resources in terms of the implied resource pattern, a set of conventions on Web Services technologies, in particular WS-Addressing. When a stateful resource participates in the implied resource pattern, it is referred to as a WS-Resource.

EGEE cannot adopt the WSRF approach currently because of its immature state. The progress of this important standardisation effort has to be tracked though and the architecture should be such that a future migration to WSRF standards is straightforward.

11.3.2 WEB SERVICE INTEROPERABILITY WS-I

The Web Services Interoperability Organisation is an open industry effort chartered to promote Web Services interoperability across platforms, applications, and programming languages [34]. The first basic WS-I document was released in April 2004, and contains simple guidelines on how to use WSDL and SOAP to be interoperable within the Web Services domain.

These specifications should be adapted by EGEE middleware as soon as the appropriate tooling is available. For legacy and early WS implementations effort should be planned to implement the WS-I guidelines.

11.4 NOTIFICATIONS

Many of the gLite services described in this document need to notify the client or other services on changes in their internal state. It is important to adopt a common mechanisms for notifications. Unfortunately, there is currently no commonly agreed upon standard for notifications in the WS community: Microsoft is proposing WS-Eventing [53] while IBM and others specified WS-Notification [54].

11.5 NETWORK ELEMENT

The current gLite architecture does not explicitly deal with the network resource but merely assumes its presence, which is usually a best efforts one. The networking activity of EGEE, JRA4, is working on the definition of a *Network Element (NE)* which will allow the management of networking resources much like the CE allows the management of computing resources. Specifically, the NE will allow reservation of network bandwidth and the definition of the desired network QoS, which can be exploited by the gLite data and job management services. In addition, statistics about the network performance will be published which again can be useful for decision making within the data and job management services. The inclusion of the NE is planned in a future revision of the gLite architecture.

11.6 DATABASE ACCESS

The current architecture as described in this document is based on the assumption that data is stored in files and that the finest level of granularity for a data item is a file. The File Catalog with its filesystem-like semantics is the manifestation of this assumption. However, many applications store their data not in files but in databases. Such applications should also profit from being able to run in a distributed Grid environment. We believe that the current architecture is flexible enough to accomodate such applications.

It has been shown that data stored in databases can be made accessible to the applications in a distributed Grid environment. Existing efforts include the EU DataGrid Spitfire project [5], the OGSA-DAI project [40] (which provides an implementation of the GGF Data Access and Integration Services working group's (DAIS-WG) [1] specifications for data access) and efforts from the industry (Oracle, IBM). Data in an existing (usually central) database may be made available to Grid jobs through such mechanisms. We will make sure that gLite is interoperable with OGSA-DAI and applications can use this mechanism to access data in databases. Still, distributing the data in a scalable, secure, controlled manner, co-locating it with the running jobs, making it writable at many sites (i.e. multi-master database replication) is a nontrivial problem that has kept the database community busy for the last two decades. Some commercial solutions exist, but none provide interoperability and platform-independence, which is necessary for Grid deployment.

We are actively involved in the DAIS-WG of GGF and will also make sure that the proper binding layers are provided to existing databases in our user communities. However, some problems might be too difficult to solve within the lifetime of this project.

12 IMPLEMENTATION CONSIDERATIONS

The Grid system realised by the services described above should allow a maximum of flexibility in service deployment, service composition, and service interoperability.

In order to achieve this, implementations of the services need to take into account the requirements discussed in Section 3. The main issues include:

Interoperability Service implementations need to be interoperable in such a way that a client may talk to different independent implementations of the same service. Following a strict SOA approach with well-defined interfaces specified in WSDL will help achieve this goal.

In addition, the Grid services need to be able to co-exist with, and leverage existing Grid infrastructures like LCG `http://cern.ch/lcg`, Grid2003 `http://www.ivdgl.org/grid2003/`, or NorduGrid `http://www.nordugird.org`. This can be achieved in developing lightweight services that only require minimal support from their deployment environment.

Service Deployment Grid services need to be easily deployable and configurable across a wide range of platforms. This goes along the lines of the goal of interoperability with existing infrastructures discussed above. In addition, several deployment scenarios need to be supported (e.g. services running together on the same physical machine, services supporting single or multiple VOs, etc.).

Service Autonomy Although the services constituting the gLite architecture are supposed to work together in a concerted way, they should be usable also in a stand-alone manner in order to be exploitable in different contexts. Ideally, if a user only requires a subset of services to achieve his task, he should not be forced to use additional services; in reality, this goal might not always be achievable and at least stubs or dummy services might be needed, however, the service design and implementation should proceed in that direction. The inverse of this argument needs to be supported as well: in certain circumstances there might be the need to include additional services into the Grid system. While a service oriented architecture in general foresees this dynamic extension, the service implementations also need to be compliant with this goal, by using dynamic discovery mechanisms, for instance.

13 CONCLUSIONS

This document presented the architecture of the EGEE Grid middleware, called *gLite* using a service oriented architecture approach. Five main logical service groups have been identified which themselves contain a set of services.

Table 2 summarises these service groups and individual services indicating the scope and enforcement of their policies. Note that this table is more detailed than the one presented in Section 2 as we have discovered the internals of high level services.

The detailed specification of these services is subject to future work and will be described in a separate design document (Deliverable DJRA1.2).

It is also worth noting that the architecture described in this document is subject to a continual evolution, based on experiences from early prototype implementations of the services described, user feedback, and the evolution of application requirements.

Service Group	Service	Scope
Security Services		
	Authentication	global
	Attribute Authority	VO
	Policy Assertion	VO, site
	Auditing	VO, site
Grid Access		
	Grid Access Service	VO
	Configuration Service	VO
Information and Monitoring		
	Producer Service	global, VO, site
	Consumer Service	global, VO, site
Job Management Services		
	CE Acceptance service	VO, site
	CE Job Controller Service	VO, site
	Workload Management Service	VO
	Job Logging & Bookkeeping Service	VO
	Job Provenance	user, VO
	Package Manager	VO
Data Services		
	Storage Element	VO, site
	File Catalog	VO
	Metadata Catalog	VO
	Replica Catalog	VO
	File Transfer Service	site
	File Placement Service	VO, site
	Data Scheduler	VO
Accounting		
	Accounting Service	global, VO, site
Site Proxy		
	Site Proxy	global, VO, site

Table 2: gLite Services and their Scope

REFERENCES

[1] GGF Data Access and Integration Services Working Group. Data Acces and Integration Services. http://forge.gridforum.org/projects/dais-rg/document/.

[2] Alfieri R. et al. VOMS, an Authorization System for Virtual Organizations. In *Grid Computing, First European Across Grids Conference*, 2004.

[3] A. Andronico, R. Barbera, et al. GENIUS: a simple and easy way to access computational and data grids. *Future Generation of Computer Systems*, 2003.

[4] Alexander Barmouta and Rajkumar Buyya. Gridbank: A grid accounting services architecture (gasa) for distributed system sharing and integration. In *Proceedings of the 17th Annual Interna-*

tional Parallel & Distributed Processing Symposium (IPDPS 2003) Workshop on Internet Computing and E-Commerce, 2003.

[5] William Bell, Diana Bosio, Wolfgang Hoschek, Peter Kunszt, Gavin McCance, and Mika Silander. Project spitfire - towards grid web service databases. In *Global Grid Forum Informational Document (GGF5)*, Edinburgh, Scotland, July 2002.

[6] P. Buncic, A.J. Peters, and P. Saiz. The AliEn system, status and perspectives. In *2003 Conference for Computing in High-Energy and Nuclear Physics (CHEP 03)*, La Jolla, California, 2003.

[7] P. Buncic, F. Rademakers, R. Jones, R. Gardner, L.A.T. Bauerdick, L. Silvestris, P. Charpentier, A. Tsaregorodtsev, D. Foster, T. Wenaus, and F. Carminati. Architectural Roadmap towards Distributed Analysis. Technical report, LHC Computing Grid Project, October 2003.

[8] Steve Burbeck. The Tao of e-business services. http://www-106.ibm.com/developerworks/webservices/library/ws-tao/.

[9] F. Carminati, P. Cerello, C. Grandi, E. Van Herwijnen, O. Smirnova, and J. Templon. Common Use Cases for a HEP Common Application Layer – HEPCAL. Technical report, LHC Computing Grid Project, 2002. http://project-lcg-gag.web.cern.ch/project-lcg-gag/LCG_GAG_Docs/HEPCAL-prime.pdf.

[10] F. Carminati and J. Templon (Editors). Common Use Cases for a HEP Common Application Layer for Analysis – HEPCAL II. http://lcg.web.cern.ch/LCG/SC2/GAG/HEPCAL-II.doc.

[11] Andy Hanushevsky et al. The SLAC Virtual Smart Card project.

[12] C. Rigney et al. RFC2865: Remote Authentication Dial In User Service (RADIUS). http://www.ietf.org/rfc/rfc2865.txt.

[13] Karl Cajkowski et. al. The WS-Resource Framework, 2004. http://www-106.ibm.com/developerworks/library/ws-resource/ws-wsrf.pdf.

[14] M. Myers et al. RFC2560: X.509 Internet Public Key Infrastructure Online Certificate Status Protocol (OCSP). http://www.ietf.org/rfc/rfc2560.txt.

[15] European Grid Authentication Policy Management Authority for e Science. http://www.eugridpma.org/.

[16] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. In *Open Grid Service Infrastructure WG, Global Grid Forum*, June 2002.

[17] Apache Software Foundation. Chainsaw. http://logging.apache.org/log4j/docs/chainsaw.html.

[18] Apache Software Foundation. log4j. http://logging.apache.org/log4j/.

[19] Apache Software Foundation. Logging Services. http://logging.apache.org/.

[20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[21] EDG Application Working Group. Joint List of Use Cases. https://edms.cern.ch/document/386184.

[22] EGEE Application Working Group. Biomedical Application Requirements. https://edms.cern.ch/file/474424.

[23] GGF Site AAA Research Group. Grid Authentication Authorization and Accounting Requirements. http://forge.gridforum.org/projects/saaa-rg/document/.

[24] The GGF Grid Storage Resource Manager Working Group.

[25] GSI: Grid Security Infrastructure. http://www.globus.org/security/overview.html.

[26] J. Howard, M. Kazar, S. Menees, D. Nichols, and M. West. Scale and performance in a distributed file system. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 1–2. ACM Press, 1987.

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

[28] Paul J. Leach and Rich Salz. UUIDs and GUIDs, February 1998.

[29] DataGrid WP1 members (G. Avellino et al.). The first deployment of workload management services on the eu datagrid testbed: feedback on design and implementation. In *2003 Conference for Computing in High-Energy and Nuclear Physics (CHEP 03)*, La Jolla, California, 2003.

[30] DataGrid WP1 members (G. Avellino et al.). The EU DataGrid Workload Management System: towards the second major release. In *2003 Conference for Computing in High-Energy and Nuclear Physics (CHEP 03)*, La Jolla, California, 2003.

[31] O. Mulmo and V. Welch. Using the Globus Toolkit(R) with Firewalls. *Clusterworld magazine*, March 2004.

[32] OASIS. eXtensible Access Control Markup Language (XACML). http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml.

[33] OASIS. Security Assertion Markup Language (SAML). http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security.

[34] The Web Services Interoperability Organization. WS-I Documents. http://www.ws-i.org/Documents.aspx.

[35] E. O'Tuathail and M. Rose. RFC3288: Using the Simple Object Access Protocol (SOAP) in Blocks Extensible Exchange Protocol (BEEP). http://www.ietf.org/rfc/rfc3288.txt.

[36] F. Pacini. JDL Attributes. DataGrid-01-TEN-0142, 2003. http://www.infn.it/workload-grid/documents.html.

[37] L. Pearlman, V. Welch, I. Foster, C. Kesselman, and S. Tuecke. A Community Authorization Service for Group Collaboration. In *Proceedings of the IEEE 3rd International Workshop on Policies for Distributed Systems and Networks*, 2002.

[38] Andreas-J. Peters, P. Saiz, and P. Buncic. Alienfs - a linux file system for the alien grid services. *ECONF*, C0303241:THAT005, 2003.

[39] R. Piro, A. Guarise, and A. Werbrouck. An economy-based accounting infrastructure for the DataGrid. In *Proc. of the 4th International Workshop on Grid Computing (Grid2003)*, Phoenix, Arizona, USA, November 2003. SC2003.

[40] OGSA-DAI project. Website. http://www.ogsadai.org.uk/index.php.

[41] UMich Kerberos PKI Project. http://www.citi.umich.edu/projects/kerb_pki/.

[42] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7)*, Chicago, IL, July 1998.

[43] IRTF AAAarch RG. RFC2904: AAA Authorization Framework. http://www.ietf.org/rfc/rfc2904.txt.

[44] P. Saiz, P. Buncic, and A.J. Peters. AliEn Resource Brokers. In *2003 Conference for Computing in High-Energy and Nuclear Physics (CHEP 03)*, La Jolla, California, 2003.

[45] David Sprott and Lawrence Wilkes. Understanding Service-Oriented Architecture. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnmaj/html/aj1soa.asp.

[46] Unified Modeling Language. http://www.uml.org/.

[47] Globus Alliance Website. http://www.globus.org/wsrf/.

[48] V. Welch, I. Foster, C. Kesselman, O. Mulmo, L. Pearlman, S. Tuecke, J. Gawor, S. Meder, and F. Siebenlist. X.509 Proxy Certificates for Dynamic Delegation. In *3rd Annual PKI R&D Workshop*, 2004.

[49] GGF OGSA WG. Open Grid Services Architecture – Glossary of Terms. https://forge.gridforum.org/projects/ogsa-wg.

[50] GGF OGSA WG. The Open grid Services Architecture, Version 1.0 (draft 016). https://forge.gridforum.org/projects/ogsa-wg.

[51] IBM Developer Works. Modeling stateful resource with Web services. http://www-106.ibm.com/developerworks/library/ws-resource/ws-modelingresources.html.

[52] IBM Developer Works. Web Services Standards. http://www-106.ibm.com/developerworks/views/webservices/standards.jsp.

[53] WS-Eventing. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/WS-Eventing.asp.

[54] WS-Notification. http://www-106.ibm.com/developerworks/library/specification/ws-notification.