The DataGrid Architecture
Version 2

German Cancio, CERN Steve M. Fisher, RAL Tim Folkes, RAL
Francesco Giacomini, INFN Wolfgang Hoschek, CERN Dave Kelsey, RAL
Brian L. Tierney, LBL/CERN

July 2, 2001

Contents

Chapter 1

Introduction

This is the second version of a document which defines the architecture of the EU DataGrid. This is a large undertaking and unlike other software projects we have worked on, some parts of the system are new to us. We are not just re-painting and re-assembling old components but planning to build something new. Consequently for now we can only describe what we see as a feasible architecture for the DataGrid. The new components will have to be prototyped - and in some cases these prototype components may have to be rejected.

This document will evolve during the lifetime of the project. It does not say what will be in any particular prototype but gives a longer view.

This version of the document has taken into account only some of the WP8-10 replies to the first 3 sets of questions posed by the ATF to the applications. It does not consider the comments recently posted by the WP8TWG to the pre-release of this document, as it was only received less than one working day ago.

We consider the work of the GGF to be very important as a genuine grassroots attempt to define Grid standards. We plan to work with the GGF both by contributing to the standardisation work and by moving towards these standards as they evolve.

1.1 Protocols vs. APIs

The system is made up of a set of GridServices. Each service is accessed via a port as is the case with httpd or ftpd. This means that a service is defined ultimately by the protocol it speaks as this defines its interface. However the normal programmer does not see the protocol itself but accesses it through a client API. In defining the system we choose to think first of the API and then of the protocol needed to communicate via that API to the service.

The division of the system into different services, their APIs and their protocols is one major ingredient of the architecture. However we have also gone inside each service a little to outline how they work and to demonstrate that they can interwork to achieve the desired functionality.

1.2 Representation of the architecture

We have chosen to model the system using UML (Universal Modelling Language) [9, 1, 10] and to make use of 4 kinds of diagram:

Class diagrams to show the various classes and their methods

Collaboration and Sequence diagrams to show how the objects interact

Use Case diagrams to illustrate a Use Case

Deployment diagrams to show the relationships between hardware and software components of the system

We have defined the APIs as Java calls. We recognise the need to provide API definitions in other languages but it is preferable to develop the model with Java in mind for a number of reasons. Java is a pure OO language which does not permit dirty tricks and as it is a high level language one does not have to worry about details of calling by address, value or reference which all serve to obscure the essentials of the design. We imagine that the client side of the code will be small - just enough to talk via the protocol, so it will probably be better to write client libraries from scratch for the different languages rather than compromising.

1.2.1 APIs in different languages

We plan to use thin clients so that they can be easily replaced by a client in a different language rather than by wrapping of one language by another. This allows an optimum interface to be developed in each language.

In considering the naming conventions for the different languages we want a solution which is consistent with the accepted style in the various languages and which does not produce excessively long names. The Java convention is to start the package name with a reversed DNS name. So the proposal below implies that we register edg.net. The domain edg.org has already been taken. Another possibility would be edg.eu.int if we can get it as eu.int is for the eu.

We will use the following conventions for naming, where in each case we consider the submit operation of a Scheduler of the workload management (WP1):

Language	Example	Notes
Java	`net.edg.workload.Scheduler.submit()`	Follow the convention of starting with a reverse domain name
C	`edg_workload_scheduler_submit()`	Pure C so no namespaces
C++	`edg::workload::Scheduler.submit()`	Use namespaces rather than prefixes
Python	`net.edg.workload.Scheduler.submit()`	Python conventions are not too clear - so we will follow the Java style

Where `workload` is a mnemonic for the package. These are, from 1 to 5: `workload`, `data`, `info`, `fabric` and `storage`.

1.3 Resources

The lowest level of GridServices are known as resources following the nomenclature defined by Foster, Kesselman and Tuecke [7] and as exemplified by the the Data Grid Reference Architecture (DGRA) [3]. A resource is normally a small layer on top of some existing software.

The two most important resources are ComputingElements (CE) and StorageElements (SE). We define a ComputingElement to be any CPU resource, whether batch or interactive that can be used by the Grid. This implies that the resource supports the Globus GSI authentication, and supports a standard Grid interface, defined below. A ComputingElement is a layer on top of a local resource management system such as a batch system. This layer hides the differences between the different underlying batch or interactive systems being used.

A StorageElement is defined to be any storage system (e.g.: a mass storage system or a disk pool) that supports the Globus GridFTP protocol and hence GSI authentication, and exposes a standard Grid interface, as defined in Section 2.3.

Making use of resources are various collective services (again using DGRA terminology). They may make use of other collective services. A GridScheduler, which makes use of services from all over the grid to decide which resources a job should use, is an example of such a service

1.4 Information and Monitoring System

Note: some of the material from this section will be moved to the services section as the appropriate services are defined

The Information and Monitoring System (IMS) will handle all information produced by the the Grid middleware and application monitoring.

The architecture is based on that of the Grid Monitoring Architecture [14] of the Global Grid Forum. The GMA as shown in Figure 1.1 consists of three components: consumers, producers and a directory service, which we prefer to refer to as a registry as it avoids any implied structure. In the GMA producers register themselves with

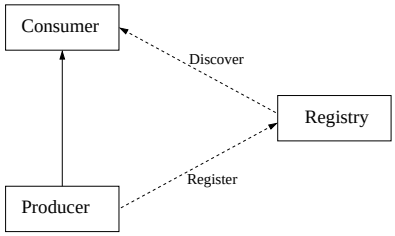

Figure 1.1: Grid Monitoring Architecture

the registry, which may itself be distributed, and describe the type and structure of information they want to make available to the Grid. Consumers can query the registry to find out what type of information is available and locate producers that provide such information. Once this information is known the consumer can contact the producer directly to obtain the relevant data. By specifying the consumer/producer protocol, and the interfaces to the registry, one can build an inter-operable Grid information service. Note that the architecture also supports joint consumer/producer components as illustrated by the component at the centre of Figure 1.2. This could gather data from several producers and make digested information available to other consumers. It might make use of an RDBMS to hold data.

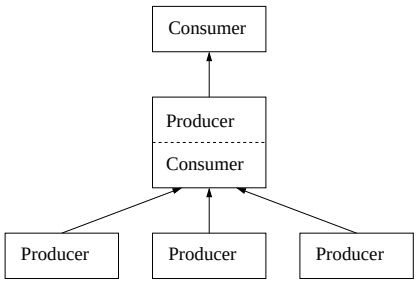

Figure 1.2: Joint Consumer Producer

The Global Grid Forum (GGF) has proposed the GMA as a monitoring architecture but we propose to use it for both information and monitoring. We plan to have a common interface to access data, whether it is *fresh* monitoring data or data from an archive. It is likely that the Relational scheme proposed to the GGF by Fisher [5] will be used and that this will also be used for registering the producers. This should be consistent with the principles of the GMA.

The API we are considering does not expose the registry. A producer simply has to announce a table name and the row(s) of a table. Behind the scenes the producer will communicate with a servlet, which will register the table name and the identity and value of any fixed attributes. Consumers can issue SQL queries against a set of supported tables (the names and attributes of which can be consulted). Behind the consumer API the query goes to a servlet which analyses the query and makes use of the registry to find suitable producers of information. In the general case queries will be sent to different producers and the servlet acting on behalf of the consumer will process the results. Queries which can be processed by a single producer can be handled efficiently, but others will result in some operations being carried out by the consumer servlet. This suggests that there will be *advantages in having Producer/Consumer/RDBMS units able to hold data which will often be joined*.

The schema information, i.e. the table descriptions, must be universally known. A RDBMS can hold both the schema and an associated list of the available producers. This would need to be replicated for scalability and reliability, but it must done in such a way as to allow producers around the world to add new tables. The producers will periodically re-announce themselves, but they will be dropped from the list of producers if they do not do so within a defined interval. When a producer registers itself as a producer of a certain table, if the table is not known it can be added to the schema. If a table is not available from any producer then its definition can be removed. This is convenient for schema evolution.

Two services will be offered to the other middleware and to end users: a Producer service 2.5 to producers of information and a Consumer service 2.6 to consumers of information. Almost all (if not all) GridServices will also act as Producers (of information) and will be able to register the various kinds of information they have on offer. For example a ComputingElement will register as a producer of information on such things as queue size, available batch capacity and suitability for interactive work. Information consumers will be able to locate suitable Producers to answer their questions.

There will be additional services to locate suitable producers, and to split queries and merge results. These will probably become first class services but for the moment they will not be published until WP3 is a little clearer about the best way to do it. In addition an archival service will be defined. This may make use of the SQLDatabaseService described in Section 2.4 or it may be preferable to package things differently.

The current information system is built on the Globus MDS. Code will be provided to publish the same information currently provided by the GRIS element of the MDS. Also, in order to facilitate migration from MDS, a gateway producer will be written to publish data from MDS.

1.5 Computing Fabric

The functionality provided by a Computing Fabric can be classified into two main categories:

- User job execution on batch and interactive CPU services - i.e. provision of the ComputingElement functionality (see Section 2.2)
- Administration of the computing fabric

The ComputingElement (CE) is the interface to computing power provided in a fabric for user job execution on batch and interactive CPU services. It handles job submission requests coming from the GRID and their execution. It also provides to the GRID information on itself and the jobs it is managing.

The administration functionalities are defined with the aim of automating the management of a computing fabric. They include the system and software installation and update of CPU farms, but also the monitoring, automated fault detection and recovery of these nodes. The administration functionalities are described in Section 3.3 on Fabric Management.

1.6 Storage Model

A Grid StorageElement (SE) is the generic name for any storage resource that includes a Grid interface (authenticated using GSI), and would be listed in a Grid ServiceIndex. The SE, like the ComputingElement (CE), is part of the Grid "fabric", as defined in the *Anatomy of the Grid* paper [7]. StorageElements will include large HSM systems like HPSS [8], Castor [2], SATSTORE[13], and ENSTORE [4], and will also include Grid managed disk pools. Note that an SE does not include CE "Working Storage" (WS), such as CE scratch disk (for example `/tmp` on a farm node), as such storage will not be directly accessible via the Grid. Working Storage is local to a CE and cannot be accessed via Grid tools from other CEs. Also, WS is volatile and only available to the job during its execution. There may be one or more SEs defined to be "local" to a CE, and this must be specified in the IMS. Data is managed in units of named files, which are grouped into named "collections" to make their handling easier and to reduce bookkeeping overhead. The mapping of files to collections is defined by the Grid applications, not by the Grid middleware.

To provide the maximum flexibility and efficiency in a Grid environment, the SE should be optimized for both local and wide-area access. This provides the most flexibility to a GridScheduler on how to schedule jobs and when and where to create replicas. For example, if the job is to be run on a computing element with an extremely high-speed network path to a remote SE, it may be more efficient for the job to do a "remote open" on the remote StorageElement than to create a local replica. Also, if a job knows that it will only need 0.5 GBytes out of the middle of a 10 GB file, then it could specify this as part of the job description, allowing a GridScheduler to determine if it is more efficient to do a remote open or to create a local replica. Whether the data is local or remote will in fact be completely transparent to the application. The same SE API, described in section 2.3, will be used for both.

There are two types of files in a Data Grid: “master” files and “replicas”. A replica is any copy of a file other than the master. The master file is owned and managed by the creator of the file, but the replicas are managed by the Grid (middleware). For example, a StorageElement may delete unused replicas to make space available for new replicas without notifying the owner of the file. The use of replicas is transparent to users; they are created as needed by the Grid middleware in order to improve overall performance of jobs. However, sites can explicitly ask for the creation of replicas locally. Initially replica files are by definition read-only; read-write implies the creation of a new master file. This is to avoid the extremely difficult synchronization problem of allowing users to write to multiple replicas of the same file. Consistency mechanisms, such as guaranteeing that when a master file is removed that all replicas are removed are described further in section 2.12. Additionally, the use of various special purpose consistency models for updating a replica and propagating the changes to the master and all other replicas are being investigated. Access control mechanisms for both master and replica files are described in section 3.4. Master files would typically be stored on a “reliable” system, (i.e. backed up), whereas a Replica does not require backups.

A simple example of Replica usage is as follows: To improve the performance of a DataGrid job to be run at site A, data in permanent storage at site B is copied to site A. This data may then be used by subsequent jobs at site A, or may be needed by jobs at site C, which has a better network connection to site A than site B. For this reason, the data should be kept at site A as long as possible. However, there is no need to store this file permanently at site A, because the file can always be retrieved from site B. The ReplicaManager, described in section 2.12, keeps track of all replica data so that the replica selection service can select the optimal physical file to use for a given job, or to request the creation of a new replica. Replica usage can be thought of as a type of long-term cache, where the data remains in the cache for use by future jobs until the cache is full, in which case the least recently used “unpinned” files are removed. StorageElement caching and pinning mechanisms and APIs are described in section 2.3.

Both master and replica file include the notion of a “lifetime”. Master files may be given a finite lifetime so that they can be deleted automatically by the system. Replicas may always be deleted by the system, but they may also be assigned a lifetime so that they are not deleted too soon. A replica lifetime might be set manually by a user who knows they will be using the same file for a series of jobs, or it could be set by the scheduler.

Replicas are currently defined in terms of files and not objects. The initial focus is on the movement of files, without specific regard for what the files contain. We realize that many users are mainly interested in objects. However, we believe we that there are well defined mechanisms to map objects to files for both Objectivity and ROOT, and that all of this will be completely transparent to the applications. However, achieving this transparency will require close interaction with the applications’ data model. In the case of most other commercial databases products, it appears that this is difficult to do efficiently, and requires additional study. Once the handling of files is well understood, further requirements analysis can extend or build on the replication paradigm to apply it to the movement of objects, structured data (such as HDF5), and segments of data from relational databases, object-oriented databases, hierarchical databases, or application-specific data management systems.

An important capability for a Grid StorageElement is the ability to “pin” a file on disk. For Master files, this means the file will not be migrated to tape and deleted from the disk pool. For Replica files, this means the file will not be purged to make room for other replicas. The Grid Scheduler (section 2.8) is responsible for pinning a file and the pinning API is discussed in section 2.3.

1.6.1 Models for ComputingElement to StorageElement Usage

There are several possible ways that a Grid CE and SE may be connected. Depending on what types of systems are available, what Grid interfaces they provide, and what networking infrastructure is in place, there are different protocols, APIs and security mechanisms that may be used. This section is an attempt to address some of these issues. The following usage models are not complete, but represent the most important combinations of components.

In the discussion below we assume the following:

- A Grid StorageElement (SE), by definition, supports GridFTP protocol and hence GSI security. Therefore it is assumed that all MSS systems in the models below have been modified to support GSI and GridFTP, or will have a GridFTP gateway in front of them.
- CE working storage (WS) is either local disk (e.g. /tmp) or looks like local disk to the application (e.g.: NFS or SAN attached disk). The Client API for working storage is always Unix I/O.

- A “local” SE means that there is a low latency (less than about 5 ms) and high bandwidth (greater than 500 Mbits/sec) network in place. Normally this means that the SE must be at the same site as the CE, but not necessarily.

Protocol Options

All data transfers to or from StorageElements will be performed using the GridFTP protocol. Applications may use the GridFTP protocol, or may use “local” protocols like NFS, if the SE is connected to the CE via NFS or a Storage Area Network (SAN). (Note: at least for the short term, the RFIO protocol will likely also be supported.)

API Options

The current plan is to develop a “GridIO” API, based on the RFIO API, which will provide applications with calls such as `GridOpen()`, `GridRead()`, `GridWrite()` and `GridLseek()`. This will provide full Unix I/O-like semantics to Grid StorageElements. If the SE is connected to the CE via NFS or a SAN, then the Unix I/O API may be used.

Security Options

For all cases where the GridFTP protocol is used, GSI security is used. This will be enhanced to use authorisation mechanisms as they become available. For all cases where local protocols are used (i.e. NFS or SAN or local disk), then only Unix UID type security is available. In the future, the new POSIX ACL interface may also be supported.

Applications that require full ACL security for each file will be required to replace their file `open()` calls with a `GridOpen()`, which will eventually provide the necessary authentication and authorization via GSI. For applications where this is not possible (for example, if the source code is not available), it will be necessary to copy the data to CE Working Storage, or to a local SE that is NFS mounted or SAN attached, so that the standard Unix open can be used. Another option for this type of application is to use the Condor Bypass system (http://www.cs.wisc.edu/condor/bypass/), which uses the Unix LD_PRELOAD mechanism, and can be used to intercept the standard Unix I/O calls and replace then with their GridIO equivalents. The potential use of Bypass needs more study.

Usage Models

Figure 1.3 shows the main models for CE–SE interaction, and the protocols, APIs and security systems for each.

In option 1, for an application that can only access the data via the standard Unix I/O API, data on the SE should be copied to the CE’s working storage where it can be accessed by the application.

Option 2 shows copying the data to a local disk pool SE. To access data on the disk pool SE, the application uses the GridIO API. If the application does not use GridIO, it can either use Bypass to remap Unix I/O to GridIO, or it can copy the file to Working Storage, where it is possible to use Unix I/O to access the file.

Option 3 shows the situation where the local disk pool SE is directly connected to the CE via NFS or a SAN, providing local file access semantics. In this environment is will be possible for applications to use the Unix I/O API, avoiding the need to modify their applications to use the GridIO library. *Of course this ability comes at the expense of losing the GSI security mechanisms, and should be considered carefully*.

In option 4 the remote SE system is CASTOR. With CASTOR (or another MSS that allows remote I/O) it is not necessary to copy the data anywhere. Files can be staged from tape to disk within the SE, and then pinned to ensure they remain on disk for the duration of the job. Applications may use either the GridIO or the RFIO API to read data directly from CASTOR. Data may also be copied to working storage and accessed using Unix I/O if necessary.

Option 5 shows the remote DB case. *(Not yet done)*.

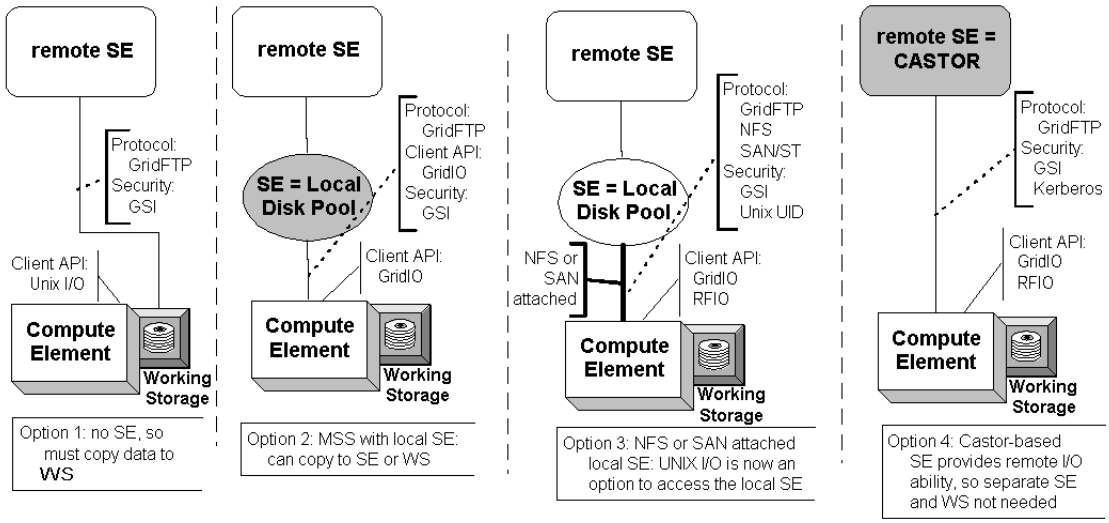

Figure 1.3: SE to CE Usage Models

1.7 File and replica identifiers in a grid environment

One of the most important DataGrid challenges is to provide convenient and efficient global file access to users. This goal can be achieved by transparently scheduling and optimizing I/O across the grid. A logical file can exist as multiple physical replicas, with each replica potentially being kept at a different SE. A protocol specific file name - the transport file name - can be derived from the physical file name. This mechanism allows multiple protocols to be supported, yet hides the protocol from the applications. Given the world wide scope of the Grid, it is essential to use LFNs and PFNs which are globally unique.

Each virtual organisation, group or individual should have as much control as possible over the structure and conventions used within their namespace. The naming scheme should be intuitive and flexible.

1.7.1 Logical File Name

A logical file is identified by a globally unique string conforming to some well defined syntax. We propose to adopt a URL-like syntax [RFC 2396] because it is well established as the open standard for convenient globally unique naming on the internet.

A logical file name consists of the string “lfn://” followed by a hostname, present in the DNS, followed by a “/” separator, after which any arbitrarily shaped application specific string can be appended. A virtual organisation may decide to use one or more hostnames.

Here are some examples for conformant logical file names:

- lfn://eo.esa.int/anything+:you*like.tex
- lfn://atlas.cern.ch/analysis/higgs/cand099.dat
- lfn://alice.cern.ch/grid/sim/ev?date=20001231&run=001&ev=123

The name does *not* tell where a file is physically stored, nor does it tell how the file can be accessed.

The hostname facilitates easy and scalable world-wide cross organisational name space partitioning. The hostname is here to say that all namespaces below it belong to (and are managed by) the owner of the hostname.

1.7.2 Physical File Name

A physical file name is used to uniquely identify a file on a given SE. We propose to adopt the URL syntax [RFC 2396] because it is a well established and flexible standard. A physical file name consists of the string “pfn://”

followed by the hostname of the SE, registered in the DNS, followed by a "/" separator, after which any directory path can be appended.

Here are some conforming examples:

- pfn://cms.cern.ch/grid/daq/triggers/2001/challenge02/ev001
- pfn://kinky.cern.ch/anything/you/like.tex
- pfn://castor001.cern.ch/whatever/ev001
- pfn://pcrd25.cern.ch/mydat

1.7.3 Transport File Name

A transport file name is used to identify how a file is to be accessed. The name contains sufficient information to allow a client to start retrieving the file stream.

The TFN allows multiple protocols to be used to access a single PFN. It is envisioned that the user will not normally use the TFN directly as the translation from the PFN will be handled by the middleware. A TFN may only be meaningful locally.

A transport file name is again a URL, using a restricted set of supported protocols (e.g. only gridftp, ftp, http, rfio and file). URL encoding for commands and parameters within a transport file name is not allowed. Here are some conforming examples:

- http://cms001.cern.ch/grid/daq/triggers-2001-challenge02-ev001
- gridftp://kinky.cern.ch/anything/you/like/or/may/not/like.tex
- file:///afs/cern.ch/user/h/hoschek/data/ev001
- ftp://castor001.cern.ch/whatever/ev001
- file:///castor/whatever/ev001
- file:///tmp/mydat

Here are some nonconforming examples. They have extra parameters encoded:

- http://cms.cern.ch/get?year=2001&kind=challenge02&name=v001
- http://cms.cern.ch/put?year=2001&kind=challenge02&name=v001

Note that LFN, PFN and TFN are unambiguously distinguishable by the scheme prefix.

1.8 Data Replication System

Data replication is one of the major issues in the DataGrid project. The Data Replication System is composed of several Grid Services, such as the ReplicaCatalog and ReplicaManager. The Data Replication System has several capabilities, including:

- Maintaining a consistent and scalable ReplicaCatalog. The ReplicaCatalog lists all files (logical and physical filenames) which are available in the Grid.
- Replication of files: this deals with the actual file transfer and the integration of logical and physical filenames into a ReplicaCatalog. We call this the ReplicaManager service.
- Synchronization of replicas (Consistency Service)
- Replica selection / when to create new replicas

- File-level access control
- Storage of replica metadata such as master flag, ACLs, lifetime and size.

A key component to the DataGrid is the ReplicaCatalog (RC), which is responsible for performing mappings from Logical File Name (LFN) to Physical File Name (PFN). It is critical that this component is designed to be extremely scalable and fault tolerant, since all other components depend on it.

To address this problem, we have designed a hierarchical distributed ReplicaCatalog system which contains two types of catalogs: leaf RCs and non-leaf RCs. We distinguish non-leaf ReplicaCatalogs and leaf ReplicaCatalogs as follows:

- Each SE is paired with a leaf RC (SE–RC) that contains all the LFN to PFN mappings for files stored in that SE only. In other words, there is one entry for each PFN on that SE.
- At the top of the tree is a catalog with all LFNs for a given VO.
- There is an arbitrary number of non-leaf RC's which contain pointers to leaf RCs or other non-leaf RC's. In other words, one can build a tree of ReplicaCatalogs, with a set of SE-RCs at the bottom of the tree. For example, a site with multiple SE's might have a non-leaf RC for all LFN's at that site. (See Figure 1.4)).

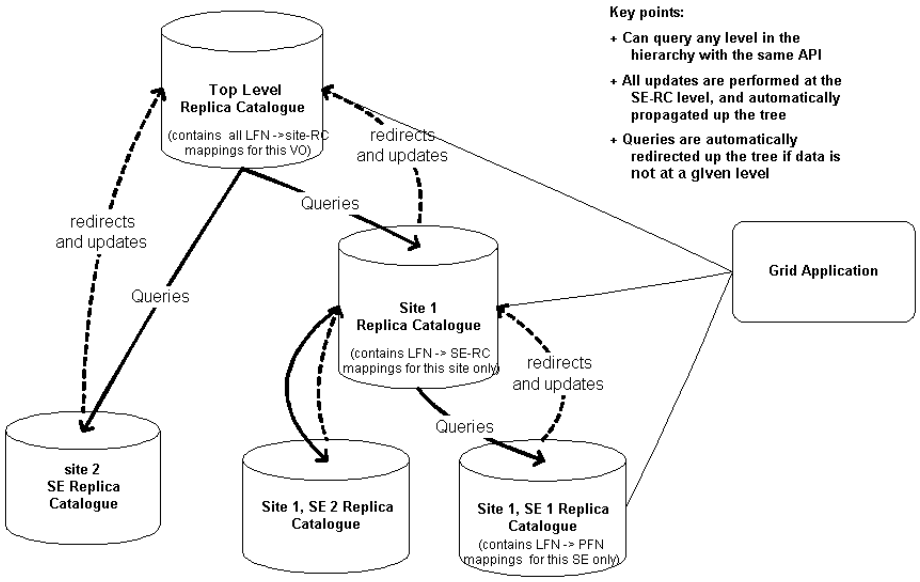

Figure 1.4: Distributed Hierarchical ReplicaCatalog

When a new file is added to the RC, it is first added to the SE-RC. This way the local catalog is always up-to-date. Periodically, say every few minutes, the SE-RC will send updates up the tree, (e.g. to the site RC, and which will periodically sent updates to the top level RC).

When doing a lookup of all PFNs for a given LFN, the Grid Scheduler (or possibly the user) can query any RC (typically it would start with the top level RC for a given Virtual Organization). This RC will automatically redirect the requests up and/or down the tree until it finds all possible PFNs. If the user knows that a replica is (or was) at a given site, they can go directly to the RC of that site or SE. If the replica is no longer there, they will be redirected back up the tree. This way the load is spread over more catalogs, relieving the top-level catalog of having to satisfy all requests.

This provides a great deal of fault tolerance. The site RC or the top level RC can be down or unreachable without effecting local operations. It also provides a high degree of scalability, as the load is distributed over a large number of RC's. The top level server should be replicated for increase fault tolerance and scalability.

More details on the RC, such as it's API, and some use cases are described in section 2.12.

1.9 The User Interface

The User Interface is expressed in terms of commands and allows three types of operations:

Session management a user can manage his own environment based on some personal or group configuration. In particular the environment initialisation includes the *grid login*, i.e. the creation of a proxy certificate to be used for all subsequent authentication operations.

Job control a user can submit and cancel jobs, list resources matching a job description, retrieve the output of a job.

Job monitoring a user can retrieve the current state of a submitted job and access some logging information.

The Job Description Language (JDL) is the formal language used to describe the characteristics of a job, such as the executable to run, input qand output files, its requirements and preferences.

The job description consists of a set of attributes and corresponding values. Any attribute can be specified, but some of them are mandatory. The attribute values are used by the Grid Scheduler to find a set of suitable resources that the job can use during its execution.

Typical attributes specify:

- the name of the executable;
- input and output data, possibly in a logical way;
- standard input, output and error;
- input and output sandboxes;
- some requirements and preferences on the environment the job needs in order to run, e.g. operating system or other software packages.

Notice how the input can be of three different types:

1. standard input;
2. data files, managed via the replication services;
3. input sandbox, which consists of relatively small files present on the user machine that need to be transferred together with the job.

Similarly for the output, which can be:

1. standard output and error;
2. data files;
3. output sandbox, a set of files which need to be moved to the user machine when the job completes.

The following example shows how a simple job description could look like (to be refined):

```
[
    Type = ``Job'';
    Executable = ``atlas.sh'';
    StdInput = ``input-file'';
    StdOutput = ``output-file'';
    StdError = ``error-file'';
    InputSandbox = ``/usr/local/atlas/atlas.sh''
    OutputSandbox = ``core'';
    Requirements = other.Type==''Machine'' && other.OpSys==''Linux'';
    IsInteractive = ``false'';
    OutputData = ``'';
    InputData = ``'';
]
```

More complex job descriptions are also possible, for example a job can consist of multiple related tasks communicating with each other using MPI.

Chapter 2

Services

2.1 Introduction

The services are listed here starting from the lower level services such as a StorageElement which offers storage facilities and moving on to the higher level services such as the GridScheduler which locates the best resources for submitting a job.

This document attempts to cover the Grid components necessary to satisfy the requirements specified in the WP8-10 Use Case Documents (see: http://cern.ch/DataGrid-WP8/Documents/Documents.html). We are expecting WP8-10 to detail a few common use cases to be included in this document. Interaction diagrams are being included in Section4 to show how the Use Case can be implemented. Currently we only have one - and that is incomplete.

2.2 ComputingElement

2.2.1 Introduction

The ComputingElement (CE) is the interface to computing power. The main functionality of a CE is to provide a uniform interface to a GRID scheduler for job submission and job control on top of the services offered by a computing fabric.

These services may be *batch oriented* using Local Resource Management Systems (LRMS) like LSF, PBS and Condor. There will also be *interactive* services, but the model for interactivity is still to be defined with WP8-10.

As the CE is a resource it will use no other GridServices except for the IMS (Information and Monitoring Service). It will use the IMS to provide information on:

- itself and
- jobs it is managing

Job submission and control

The CE will handle requests for job execution and job control operations coming from the Grid.

The main functionalities include:

- fabric (or local) authorization for grid requests
- mapping between grid credentials and local credentials
- a fabric Resource Management System (RMS) which translates the GRID job description into the local job description language and manages the local fabric batch and interactive services
- support connections from individual farm nodes to locations outside the fabric

Upon a Grid job submission to a CE, the CE checks the local authorisation and maps grid user credentials to local user credentials (eg. uid/gid, kerberos tokens). This mapping can be done statically or using dynamic account creation.

Once this has been accomplished the job is passed to the fabric's Resource Management System (RMS). The RMS sits on top of the fabric's Local Resource Management Systems (LRMS), like PBS and LSF. The RMS is *not* a GridService.

The main task of the RMS is to maintain control over the fabric's farm resources and the efficient scheduling and execution of both user jobs and administrative tasks (see section 3.3). It sits on top of the different batch and interactive services a fabric offers. It will offer *virtual queues* for load balancing. The virtual queues will not only distribute their jobs between physical LRMS queues but may also keep a backup until a job has executed in case the LRMS queue should be lost because of some system failure affecting the LRMS.

The RMS may offer enhancements on the functionality of the underlying LRMS, like:

- scheduling strategies (ie. FCFS, Backfill, shortest-jobfirst, longest-jobfirst, deadline scheduling)
- limiting resources a job can use
- resource reservation
- job monitoring
- job dependencies and chaining within a fabric
- control of what happens when a job crashes due to system failure
- local accounting

Job failure

If a job crashes due to some kind of system failure it can be rescheduled by the RMS in which case it could be run on a different LRMS queue. It could be that a job parameter specifies that if a job is not completed by some calendar time it should not be run and the RMS would then report a failure to the CE. A job parameter might also specify unconditionally failing a crashed job and reporting it to the CE or to specify that partially created output files should be preserved.

Job status information

The CE will generate a jobID which the can be used as a handle to ask about information regarding that job.

The Job status information could be made available via the IMS, by returning this information after an explicit query, or via a callback mechanism back directly to the user or Grid Scheduler (see section 2.8 after a job status change. This is yet to be clarified.

The job information may include:

- job status (ie. started, suspended, scheduled, running, failed, done)
- job queue information (length and/or estimated job start time)
- job resource usage (i.e. times, memory, swap, number of processes)

Job monitoring

Jobs which publish information, like output and debug/log files and streams) act as IMS producers in the usual way. A suitable mechanism will be devised to make the information available on the Grid so that consumers may use it. This may allow the option of archiving the information.

Accounting

The RMS will generate local fabric accounting information which will be stored for internal use and for quota management. A subset of this information could be made available for Grid wide accounting services.

CE information

The CE status will be made available via the IMS. Most of this information will come from the RMS and subsidiary systems. Information will be produced from the lowest levels and aggregated by suitable elements which act both as consumers of simple information and producers of derived information. The highest level of aggregated information will be used by the GridScheduler.

Application Environment

In order to run a job, a certain *Application Environment* is required. An Application Environment is defined by a set of software components which are required for a specific job type to run. This environment is defined by the application WPs. It may include application specific packages (eg. ATLAS analysis software v.X.Y), more generic packages (eg. Objectivity 5.2, ROOT 3.01), GNU tools (eg. gcc-3.01), system components (eg. Kernel 2.4.4, glibc 2.1.3).

Information about availability of application environments on a farm will be published in the IMS. The type, format and granularity of this information is being negociated with WP8-10.

By default and for efficiency, the Application Environment will have to be preinstalled on the fabric (see section 3.3).

As a second possibility, executables can also be staged together on job submission.

A third possibility (setting up a grid wide dynamic software service) would require further investigation for the needs and feasibility of the SoftwareRepositoryService in section 2.13.

Communication between jobs

Enabling outgoing/incoming IP traffic from fabric nodes is a per site policy decision (which is published in the IMS). No assumptions on the visibility of computing nodes from the external network is made for several reasons, including for example, security policies and IPv4 address space shortcomings.

In the case that nodes don't have external connectivity, the need and feasibility for a subsystem which would support declared connections for wide area communication mechanisms between CE's is being investigated.

This subsystem would map on request declared connections from individual jobs on fabric nodes to the outside Grid, and to other fabric nodes. This may be neccessary when using eg. MPI between CE's with restricted external connectivity.

2.2.2 Class and Object Diagrams

To be defined.

2.2.3 API

It is yet too soon to define an API and/or object diagrams as the user requirements on functionality have to be considered first. The ComputingElement will include methods for job submission and control as described above.

- submitJob (Job:JDL): JobID
- getJobInfo (id:JobID): JobInfo
- CancelJob (id:JobID): result

In addition the CE is a producer of many different kinds of information on fabric status and job information (as described above) which will be published via the IMS, including among others:

- List and types of available resources (eg. queues)
- Resource boundaries (eg. minimal available temporary working storage, maximum CPU time, maximal running jobs)
- Currrent resource status (eg. current running jobs, total jobs)
- Resource availability (eg. time windows where the resource is up)
- Installed application environment (see above)
- Attached Storage Elements

2.2.4 Protocols

To be defined. The GRAM protocol from the Globus toolkit will be considered.

2.2.5 Services needed

- IMS

2.2.6 Constraints and assumptions

- No assumptions should be made in terms of the existence of home directories or shared file systems. UIDs and home directories may be volatile - created on the fly and removed after job completion. A script provided by an application environment (see above) can be called by a job script for setting up a job environment per application class.
- Non Grid jobs may share some of the physical resources.
- Users must be uniquely identifiable to allow for resource accounting.

2.2.7 Open issues

- The full implications of supporting interactive work at a CE have yet to be understood. We need here a good and common understanding of what “interactivity” means for the application WPs.
- Job priorities and resource reservations. Support for both reservation mechanisms and priorities have been expressed by the application WPs. What kind of priorities are requested, and how reservations can be made compatible with priorities is to be clarified in the application’s requirements.
- Job checkpointing and job migration. What is the model and the specific requirements of the application WPs.

2.3 StorageElement

This section is about to be revised significantly. In particular it will offer one rather than three interfaces

2.3.1 Introduction

The StorageElement (SE) is the Grid interface to Mass Storage, be this tape or disk storage systems. It is a layer that sits above existing Storage systems and acts as the interface to the Grid middleware. For sites that only have disk storage the SE will provide a disk management system to manage the data. (Data in this case being the collection of bits being stored, ASCII text or database).

The SE will have three interfaces. The first will be a data access interface. This will provide the basic data access methods such as get, put and delete. This interface must be provided for a SE. The second two interfaces are optional, but provide further functionality. One will provide record level access to the data, such as open and close, read and write. The third interface provides a management interface to allow for the allocation of space for the data (to guarantee a read/write to grid storage will not fail due to lack of space), pinning and filename generation. If the SE is a disk-pool which is full, the allocate call may trigger the deletion of old data to make room for the new allocation. This interface will allow attributes to be set for the stored data such as its life-time on grid storage. Data that are important (i.e master copy) can be given an infinate lifetime. To emulate scratch storage data could be stored with a finite (i.e. one month) life time, and be available for following jobs. The system would remove the data after this time. Other attributes such as whether the data is a master copy and to flag the data as important and needs to be stored on secure storage (i.e. disks backed up/raid, tape copies stored in fire safes. These copies are *not* intended to allow the user/system to recover from accidental deletion, but from media failure.) This implies that theSE must publish the fact that it can support secure storage to the grid.

Using these interfaces, then more complex data storage models discussed else-where can be constructed.

2.3.2 Class and Object Diagrams

2.3.3 Protocols

GridFTP will be used as one of the WAN protocols.

2.3.4 Services needed

- The Information and Monitoring Service

2.3.5 Constraints and assumptions

2.3.6 Open issues

- How are errors handled. Do we just return an error or something meaningful that can be interpreted by the replica manager or higher level scheduler and acted on
- How will the Grid know that a SE and CE are local to each other?

2.4 SQLDatabaseService

2.4.1 Introduction

The SQLDatabaseService allows the scalable and efficient storage, retrieval and query of data held in any type of local or remote RDBMS. It is expected that this service will be used for meta data. The core functionality is SQL insert, delete, update and query. The functionality can be invoked from a command line tool, a web browser, as well as from a programming language API. A well defined language, platform and RDBMS neutral network protocol between client and server is used. At the highest level no programming skill is required at all in order to talk to the service. At the lowest level, rich customization and data transformation is programmable.

The overall architecture of the SQLDatabaseService is a classic loosely coupled 3-tier model on the web, as for example, exemplified by Google, amazon.com or the CERN web phonebook: HTTP clients ask server side applications to execute requests which are URL parameter encoded or XML formatted. The server side application defines the logic of the request and converts it to SQL (typically by substitution arguments in a template SQL query). The SQL query is dispatched to a SQL database backend tier for data storage and retrieval. Finally the server side logic trivially converts the SQL results set to canonical XML and ships it back to the client application, which goes about processing it in any desired way. If the client is a web browser, then one additional XSLT pipeline step on the server side transforms XML into HTML. This is a very popular architecture for problems of the given nature. In fact, these days it is hard to find organisations choosing any other approach.

The service uses the following mechanism in the respective domain.

- Network transport protocol for client server communication: HTTPS. Reason: A reliable request-response protocol is sufficient. https allows to leverage the massive existing infrastructure of flexible, robust & scalable, easy to use http based tools. Vendor and programming language lock-in is avoided. Buggy, slow, hard-to-deploy or otherwise inappropriate components can easily be exchanged with alternatives without effect on compatibility and interoperability.
- Data exchange format between loosely coupled components: XML. Reason: XML is very flexible, easy to use, standardized, language & platform independent, and widely supported.
- Naming: URL + tablename. A SQL table is globally unique identified by the database instance URL followed by the name of the table.
- Query language: SQL. Reason: SQL is standardized, powerful, highly efficient. It only exposes high level conceptual model and hides physical model so that physical model can be evolved and optimized without breaking any client code
- Database: any RDBMS. Reason: Relational DBs are low risk solutions which are well understood, standardized, highly efficient, robust, scalable. A large range of open source and commercial implementations are available.
- Security: https (http over SSL/TLS), X.509 Certificates (GSI compatible). Reason: Simply the only viable option

2.4.2 Interaction Diagram

Figure 2.1 depicts an interaction diagram of a complete request-response call chain.

As an example, if a client wants to retrieve the metadata associated with all known logical file names, it issues a http GET to the following URL:

```
http://sql.cern.ch/getLogicalFileMetaData?catalog='RCcatalog'
```

The SQLDBservice has a number of query template files defined, one of which specifies the appropriate SQL query:

```
<xsql:query xmlns:xsql="urn:oracle-xsql" connection = "myconnection"
    select * from {@catalog)}
</xsql:query>
```

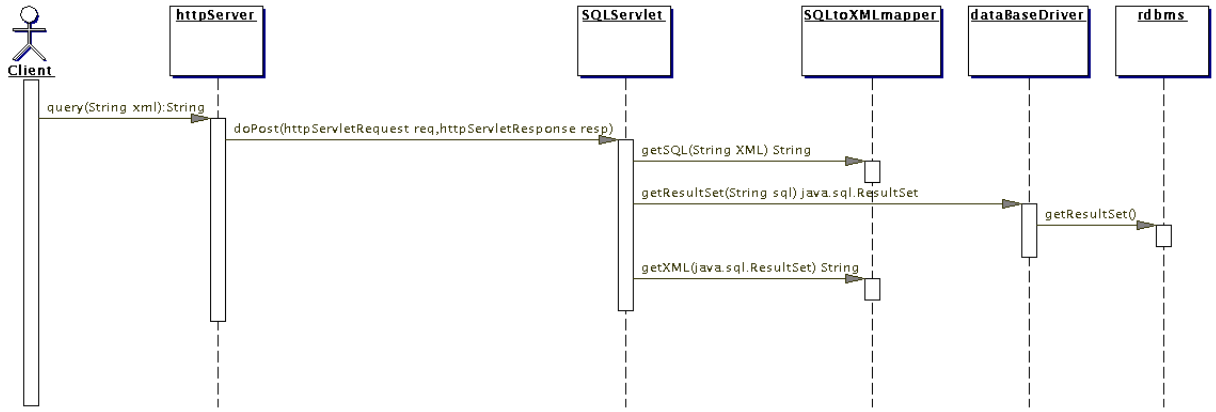

Figure 2.1: Interactions of SQLDataBaseService

The query parameter ”RCCatalog” is substituted into the template. The full query is executed and the result set returned to the client as canonical XML:

```
<ROWSET>
    <ROW>
        <lfn> host1.cern.ch/somepath/file1 </lfn>
        <size> 10000000 </size>
    </ROW>
    <ROW>
        <lfn> host2.cern.ch/somepath/file2 </lfn>
        <size> 50000000 </size>
    </ROW>
</ROWSET>
```

Note that in canonical XML a SQL table corresponds to a ROWSET element, a SQL row corresponds to a nested ROW element, and a SQL column is mapped to a nested tag with the same name, filled with the value of the column.

As a second example, let us assume a client wants to insert logical file name data into a SQL table. The client issues a http POST to the following URL:

```
http://sql.cern.ch/myinsert
```

with the body of the message holding the data to be inserted:

```
<ROWSET>
    <ROW>
        <lfn> host1.cern.ch/somepath/file1 </lfn>
        <size> 10000000 </size>
    </ROW>
    <ROW>
        <lfn> host2.cern.ch/somepath/file2 </lfn>
        <size> 50000000 </size>
    </ROW>
</ROWSET>
```

A query template file for the SQLDBservice defines the appropriate insert command:

```
<xsql:insert-request xmlns:xsql="urn:oracle-xsql"
    connection = "demo"
    table = "RCcatalog"
    transform="trans.xsl"
</xsql:insert-request>
```

2.4.3 Class and Object Diagram

The components involved include the following:

- Client: browser, commandline, user appl. (e.g. wget, netscape)
- HTTP server and servlet engine: Apache, Tomcat, modjk for load balancing
- Servlet: SQLServlet + Globus CoG
- SQL to XML mapper: XSQL et al
- Database driver and RDBMS: JDBC + Oracle, MySQL, DB2, etc.

2.4.4 API

- doHTTPGet(URL url)
- doHTTPPost(URL url, String body)

2.4.5 Protocols

https, X.509 (GSI compatible)

2.4.6 Services needed

None.

2.4.7 Constraints and assumptions

A scalable high availability deployment scenario of a single SQLDatabaseService is shown in Figure 2.2 The setup has N concurrent clients, M http servers, O runtime engines, and P RDBMS instances. Each of these can (but need not) run on a different box. Each unit can run in Q processes and R threads. Improved performance is achieved by transparent load balancing over all these levels, as well as thread, connection and transaction pooling.

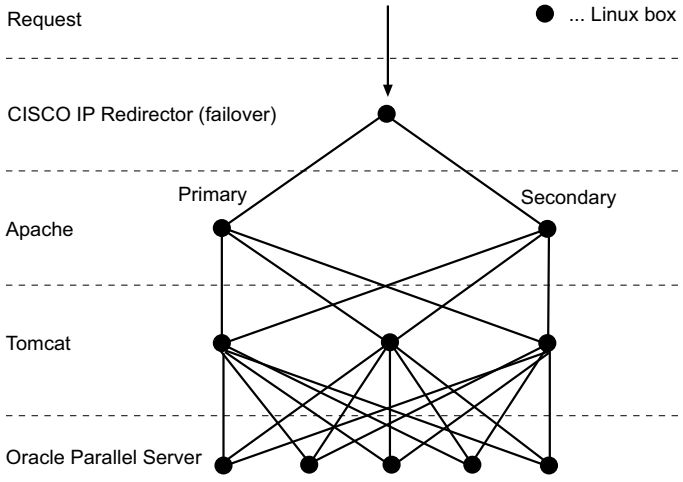

Figure 2.2: Scalable high availability deployment of SQLDatabaseService

2.4.8 Open issues

None

2.5 Producer

2.5.1 Introduction

The producer is the means of publishing to various producers (see section 2.6 for details of the consumer and section 1.4 for an introduction to the information and monitoring system). Information is published as rows of SQL tables. A producer can publish rows from a single table, and so is instantiated with a description of the "table" from which it is publishing information. Its constructor may take a SQL CREATE TABLE statement to describe the roes of the tables to be published and it is by invoking the insert method with an SQL INSERT statement as an argument. The producer object holds little code but communicates with a nearby servlet which ensures that the table structure is the the same if the table already exists, and otherwise looks after the registration of the schema and of the producer.

2.5.2 Class and Object Diagrams

Figure 2.3 shows the Producer class. A producer is instantiated with the description of the information it has to

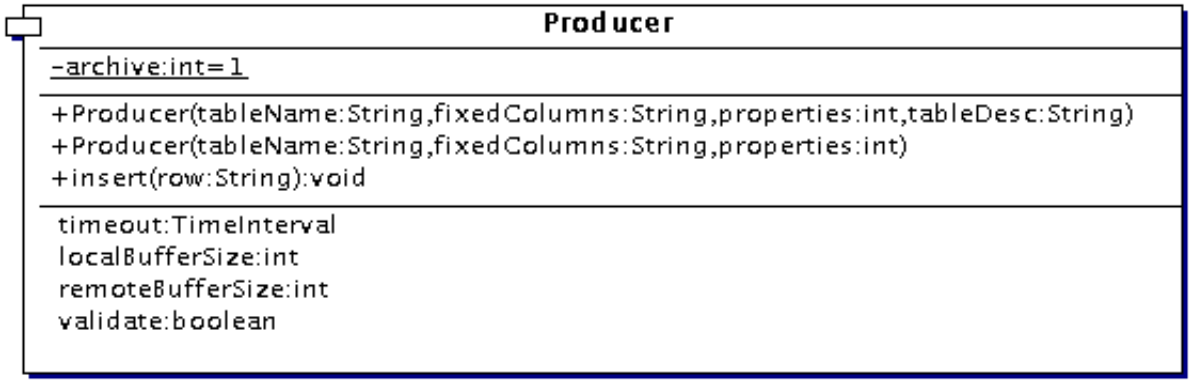

Figure 2.3: Producer API

offer. The first argument is the table name. The second is a character string describing the columns which are fixed and their values. This is a comma separated list of columnName = value pairs. The last argument to the constructor is a String describing the tables to be published. This string takes the form of an SQL `CREATE TABLE` statement. This last argument is optional - if the table is already known within the schema.

To publish data, the insert method is invoked. This takes a normal SQL `INSERT` statement. If the timestamp is not set then the producer will derive a time stamp from the system time. The localBufferSize property controls how many tuples of data are cached before a transfer is made. The timeout property allows the person publishing information via a producer to request that tuples are transmitted before the buffer is full. The remoteBufferSize controls how many rows will be held by the producer servlet if one of its consumers is slow.

2.5.3 Security

GSS/GSI

2.5.4 Protocols

The consumer/producer protocol is expected to be based on XML over http(s). Communication between the producer and servlet(s) acting on its behalf will also be based on XML over http(s).

2.5.5 Services needed

- Not yet known

2.5.6 Constraints and assumptions

- That the GMA architecture is well suited

2.5.7 Open issues

- Many uncertainties as we have no prototype.

2.6 Consumer

2.6.1 Introduction

The consumer is the means of accessing information made available by various producers (see section 2.5 for details of the producer and section 1.4 for an introduction to the information and monitoring system). A consumer handles a single query, expressed as an SQL SELECT statement, and supports either data streamed to you or executed one at a time. The consumer talks to a servlet (which is where information is really streamed to) which should be close (in network terms) to the process requesting the information. This servlet (making use of other servlets) locates the best source(s) of information, and makes a temporary connection to a producer servlet for a query to be executed once, or makes a permanent connection to allow data to be streamed. The query may need to be split into sub queries and the results from multiple producer servlets combined - this is all coordinated by the consumer servlet with which the consumer object in the users program communicates.

2.6.2 Class and Object Diagrams

Figure 2.4 shows the Consumer classes, API. The consumer is created with a String representing the SQL Query

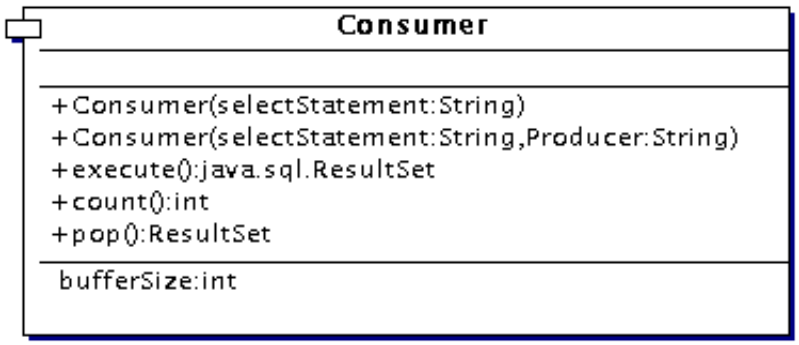

Figure 2.4: Consumer API

you wish to execute. The second constructor allows the specification of the URL of the producer to connect to. You can then either execute a query which returns (in the case of Java) a `java.sql.ResultSet` or you can set the `bufferSize` to be non-zero which allows the information to be streamed in, after which you can use `count` to see how many records are available and `pop` to get a record - again as a java.sql.ResultSet.

2.6.3 Security

GSS/GSI

2.6.4 Protocols

The consumer/producer protocol is expected to be based on XML over http(s). Communication between the consumer and the servlet(s) acting on its behalf will also be based on XML over http(s)

2.6.5 Services needed

- Not yet known

2.6.6 Constraints and assumptions

- That the GMA architecture is well suited
- That query splitting and result combination is practical

2.6.7 Open issues

- Many uncertainties as we have no prototype.

2.7 ServiceIndex

2.7.1 Introduction

The ServiceIndex allows for the construction of a distributed “web of services”. Services can register themselves with one or more indices. An index can itself register with one or more other indices. The distributed graph of services can be traversed to enable searching. This is done by higher level services outside the scope of the ServiceIndex, such as directory services, search engines and brokers.

With each service a ”Service Description” is associated. The service description contains the information a client needs in order to be able to effectively communicate with the service. Such descriptions are defined in XML. All descriptions inherit from a standardized and minimalistic base XML schema. Therefore, the base schema can be extended in a type safe manner.

Service registrations automatically expire unless periodically renewed.

The following example illustrates a sample service description of a replica manager. It defines the mandatory information such as the URL and type of service, the Virtual Organization it belongs to, as well as details about the protocol it speaks. In addition optional high level information about replica catalog contents is given so that clients can filter and exclude services not interesting to their goals. Note that the example does not claim to well specify the layout which will finally be adopted for replica managers. The final layout is likely to look different in significant ways.

```
<service>
   <mandatory>
      <URL>x-gridrep://eff555.cern.ch:55555</URL>
      <type> replica-manager </type>
      <owner> atlas </owner>
      <email> atlassupport@cern.ch </email>
      <protocol version = "1.2"
         <authentication mechanism = "GSI"/>
         <authentication mechanism = "CRAM-MD5"/>
         <authentication mechanism = "OTP"/>
      </protocol>
   </mandatory>

   <optional>
      <catalog>
         <type> atlas-trigger-studies </type>
         <runs>
            id-from   = "10000"
            id-to     = "20000"
            date-from = "2000-12-08"
            date-to   = "2001-02-20"
            status    = "disk resident"
            size      = "2 TB"
         </runs>
      </catalog>
   </optional>
</service>
```

2.7.2 Class and Object Diagram

2.7.3 API

- registerChild(ServiceDescriptionXML)
- registerParent(ServiceDescriptionXML)
- getChildren() : List of ServiceDescriptionXML

- getParents() : List of ServiceDescriptionXML

2.7.4 Protocols

https, X.509 and user/password auth.

2.7.5 Services needed

SQLDatabaseService.

2.7.6 Constraints and assumptions

Some higher level broker will provide advanced search capabilities. This service does not provide search capabilities, because such capability is domain specific, hence better implemented by some higher level domain specific search service.

2.7.7 Open issues

None

2.8 GridScheduler

2.8.1 Introduction

The purpose of the GridScheduler is to submit a given job to a suitable ComputingElement where the job can run and access all the resources it requires, e.g. data access. If the job fails for reasons which are independent of the job, it is rescheduled. To guarantee this an appropriate job monitoring has to be performed.

A job is described according to the Job Description Language syntax. A job description must contain enough information to allow a GridScheduler to match it against the resources available on the Grid.

A typical job description would include:

- the name of the executable;
- the logical specification of input and output data;
- the specification of standard input, output and error;
- the specification of the input and output sandboxes;
- some requirements on the environment the job needs to be able to run, e.g. operating system or other software packages.

The Job Description Language also allows more complex job descriptions, for example consisting of multiple tasks with some dependencies between them. Such a possibility requires the capability for the GridScheduler to co-allocate and/or reserve in advance resources, especially in the case where the tasks need to communicate with each other, e.g. via MPI.

Scheduling decisions are based on several criteria, including but not limited to:

- location of data;
- authorisation constraints;
- accounting information;
- status of the available resources;
- requirements and preferences expressed by the user.

2.8.2 Class and Object Diagrams

2.8.3 API

The API is being defined, but it will include the following:

- Job submission

  ```
  GridScheduler::submit(Job j);
  ```

- Job cancellation

  ```
  GridScheduler::cancel(Job j);
  ```

- Cancel all jobs submitted by a user

  ```
  GridScheduler::cancel_all(String user);
  ```

- Retrieve output sandbox

  ```
  GridScheduler::get_output_sandbox(Job j, String path);
  ```

- Find a list of Computing Elements matching the job submission request

  ```
  GridScheduler::list_matches(Job j);
  ```

2.8.4 Protocols

Not yet defined, but GRAM is a possibility.

2.8.5 Services needed

To achieve its task the GridScheduler needs support from many other services. Identified dependencies on other services include:

- Logging and Bookkeeping Service, for:
  - storing logging and bookkeeping information;
  - retrieving job status information
- GridServiceIndex, for:
  - registration of the GridScheduler
  - search of available services, including Replication Services, Computing Elements and Storage Elements
- Replication Services, for:
  - data transfers
  - conversion from LFNs to PFNs
  - registration/allocation of a new LFN
- ComputingElement, for:
  - submission and monitoring of jobs
- StorageElement, for:
  - allocating physical file names
  - allocating storage
  - pinning files
- Information and Monitoring system for:
  - obtaining information about ComputingElements, StorageElements and the network

2.8.6 Security

Acceptance of a user request is authenticated using user proxy credentials and the corresponding transaction with the User Interface is mutually authenticated using server credentials.

A GridScheduler may need to act on behalf of a user during the processing of a request, e.g when it submits a job to a Computing Element. Moreover this need may last for a long period of time, e.g. at least until the submitted job has finished. To allow a GridScheduler to act on a user's behalf two solution are envisaged:

1. a user provides a GridScheduler with credentials that will be valid for a long enough period of time;
2. a mechanism exists to renew expired credentials.

Additional interactions with other grid services are mutually authenticated based on host credentials.

2.8.7 Constraints and assumptions

None at the moment.

2.8.8 Open issues

- How many schedulers?
- Personal or community schedulers or both?
- Security framework.

2.9 Logging and Bookkeeping

2.9.1 Introduction

The Logging and Bookkeeping Service is used to store information about the scheduling system and the jobs that pass through it.

Bookkeeping information concern user jobs, such as job description (in JDL), job status, resource consumption and possibly user-defined data. These information are meant to exist only for the lifetime of a job.

Logging information concern the scheduling system and are mainly used for debugging, auditing and statistics purposes.

Some data may be stored both as bookkeeping and as logging information, but the use of that data would be different in the two cases.

The Logging and Bookkeeping Service can also be seen as the repository of the status of one or more Grid Schedulers. In this way only the Logging and Bookeeping Service needs to be highly reliable and available, whereas no such constraint is required for Grid Schedulers.

This facility will probably make use of the SQLDataBaseService and the IMS.

2.9.2 Job state machine

A job during its lifetime goes through a number of states:

SUBMITTED the job has been submitted by the user to the User Interface

WAITING the job has been received by the Grid Scheduler

READY a Computing Element best-matching job requirements has been selected

SCHEDULED the job has been received by a Computing Element

RUNNING the job is running on a Computing Element

CHECKPOINTED the job has been suspended and checkpointed on a Computing Element

DONE the job has completed

ABORTED the job has been terminated

CLEARED output files have been retrieved by the user, the job is removed from the bookkeeping database.

The above states and the allowed transitions between them are shown in Figure 2.5.

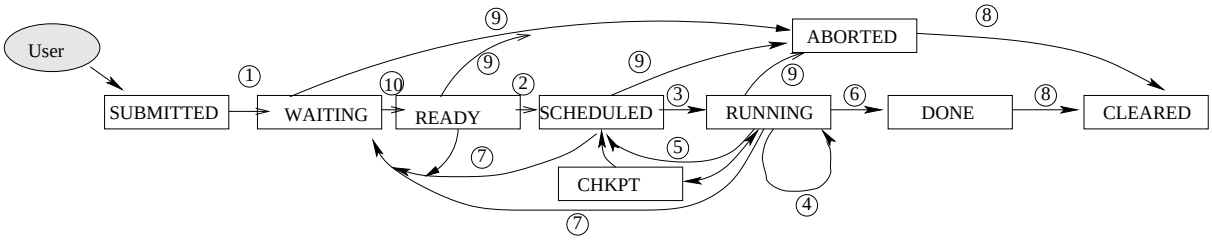

Figure 2.5: Job state machine

State changes are triggered by *events* generated by User Interfaces, Grid Schedulers and Computing Elements. Events represent the information that is actually stored in the Logging and Bookkeeping Service. Job status is usually not stored directly but can be inferred from other stored information.

Information about some states may not be available, e.g. the CHECKPOINTED state is not detectable if there is no support for it by the Local Resource Management System behind the Computing Element where the job runs.

2.9.3 Job identifier

At grid level a job is referenced by a unique identifier. Such identifier is generated by the User Interface, the reason for this being that the User Interface is the first entity to log some information (i.e. the SUBMITTED state).

The exact form of the string is up to the User Interface. All the other grid components should handle the string literally.

In order to easily associate a job to the Logging and Bookkeeping Service that holds its information, the job identifier could contain the contact string of such service. For similar reasons the identifier could also contain the contact string of the Grid Scheduler used for the job submission.

2.9.4 Class and Object Diagrams

2.9.5 API

Two APIs are being defined, one to write into the Logging and Bookkeeping Service and one to access it.

They are presented below with a C syntax. A C++ syntax is under investigation.

Producer API

The producer API contains a single function.

```
int dgLogEvent(char* jobId, char* source, dgLBEventCode event,
               int level, char* format, ...);
```

`jobId` is the grid job identifier.

`source` is the identifier of the component that generated the event.

`event` is the type of event to be logged.

`level` is the level of the event according to the ULM draft (reference needed here), e.g. Debug, Usage, System.

`format` is a `printf()`-like format string.

`...` is the list of arguments according to `format`.

Consumer API

The Consumer API contains functions to:

- manage a job identifier;
- handle events;
- manage the connection to a Logging and Bookkeeping server;
- query a Logging and Bookkeeping Service;
- handle errors;

2.9.6 Protocols

Being defined.

2.9.7 Services needed

The Logging and Bookkeeping Service will probably be implemented using:

- an SQLDatabaseService;
- the IMS.

2.9.8 Security

All commands accepted by the Logging and Bookkeepting Service are authenticated using user proxy credentials. The corresponding transactions with User Interfaces, Grid Schedulers and Computing Elements are mutually authenticated using host credentials.

Users are allowed to monitor at least their own jobs. Further authorization is under discussion.

2.9.9 Constraints and assumptions

2.9.10 Open issues

- Evaluate the implementation on top of SQLDataBaseService and IMS.

2.10 FileCopier

2.10.1 Introduction

The task of the FileCopier is to efficiently and securely copy files from one Grid location to another one, similar to FTP. This service enables efficient copying of files across the grid. It is a low level service with a simple interface and no smart behaviour. Note that it is the ReplicaManager which uses the FileCopier and updates ReplicaCatalogs.

- copyFile(TransportFileName source, TransportFileName destination):Status
  This function also allows for third party transfer.
- setTransferParameters(timeout, parallel, striped, buffersize, restartOnFailure)

2.10.2 Protocols

The GridFTP protocol will be used.

2.10.3 Services needed

- StorageElement: for holding physical files
- Monitoring, Accounting and Authorisation

2.10.4 Constraints and assumptions

None.

2.10.5 Open issues

None

2.11 ReplicaCatalog

2.11.1 Introduction

The main purpose of the ReplicaCatalog is to provide a mapping of a logical file name to one or more physical file names. For each logical file name, additional file meta information (file attributes) can be added. Furthermore, the location of a replica can then be used together with other information services to obtain the cost for accessing single replicas and creating replicas. This service thus enables storage and retrieval of information about logical files and physical files, as well as associated metadata (such as file size, timestamp, owner, etc.). The replica catalog service can be seen as the database backend tier of the replica manager. A replica catalog contains zero or more logical files, and each logical file contains zero or more physical files. The catalog imposes structure on top of physical files, but does not create, delete or read physical files. It can be implemented on top of an RDBMS, ODBMS or LDAP as backend.

A ubiquitous remote method invocation protocol (using XML over https or URL encoded https) encapsulates and shields client applications from the details of the underlying backend storage technology. Thus, different backend implementations of the Replica Catalog API (e.g. using a RDBMS, ODBMS or LDAP) can be used without introducing interoperability problems.

There are several ways to implement a replica catalogue based on the amount of distribution of replica catalogue data. We can identify two extreme cases for catalogue distribution: a centralised approach and a distributed approach:

- **Central replica catalogue service:** A centralized service and thus a central data store stores all catalogue information and is contacted by all applications that need to map a logical to a physical filename. We assume that this is the most straight forward implementation of a catalogue with a single catalogue server and several catalogue clients. *Pros*: No synchronization problem, because all files are kept in one catalogue. *Cons*: Less scalable, potential bottleneck due to WAN latency and high traffic. Catalog itself needs to be replicated to avoid outages and network problems. No separation of administrative responsibility.
- **Distributed replica catalogue service:** Each virtual organisation's site or Storage Element has a co-located replica catalog which keeps a list of locally available files. *Pros*: Very scalable due too partitioning of a large problem into many smaller problems. Easy to maintain under locally administration. *Cons*: Difficult to synchronize, difficult to find a file efficiently. Clients 'somehow' need to know about other replica catalogs if they want files not locally available.

  We assume that replica catalog sites do not need to be synchronised immediately, but only within a specified timeframe of, for example, minutes or hours (relaxed consistency). However, convergence must be achieved eventually.

For scalability and manageability reasons, we propose an approach that is a hybrid version of these two extremes. We define a hierarchical replica catalogue system with interconnected parent and child replica catalogues, forming a distributed tree or graph (see Figure 2.6). The structure can be traversed. We can distinguish non-leaf Replica Catalogs and leaf Replica Catalogs:

- A leaf Replica Catalog stores logical file name (LFN) to physical file name (PFN) mappings (one-to-many) for each locally stored physical replica, plus file attributes for its LFNs and PFNs. File atttributes include file size, time-step, check sum, creator, master/replica, owner etc.
- A non-leaf Replica Catalog stores logical file name (LFN) to Replica Catalog URL mappings (one-to-many) for each LFN with at least one physical replica in its subtree.

In other words, non-leaf replica catalogues do not have physical file information but only store the URL of child replica catalogues. A non-leaf catalogue redirects requests for a physical filename to children, which may in turn redirect until physical file information is found in a leaf Replica Catalog. The additional redirection lookup steps should be transparent to the end user. Each Replica Catalog maintains a local and autonomous view of its subtree.

Whether leaf or not, all Replica Catalogs speak the same lookup protocol and provide exactly the same lookup API. Consequently, any hierarchical or graph-oriented layering is possible. One possible structure is as follows. With each Storage Element multiple leaf replica catalogues can be co-located (one per virtual organisation). We refer to them as Storage Element-Replica Catalogs (SE-RC). We anticipate that a site will host several Storage

Elements and virtual organisations. Hence a site may integrate all its Storage Elements under one Site Replica Catalog or under one Site Replica Catalog per virtual organisation. Small sites may choose to omit a site Replica Catalog.

As an example consider one potential structure for the CMS experiment (see Figure 2.7): On each Storage Element a Storage Element-Replica Catalog is started up just for CMS. CMS also decides to configure an intermediate Replica Catalog at each Tier 1 site (INFN, RAL, etc). Plus, there will be a single, global Root Replica Catalog at CERN.

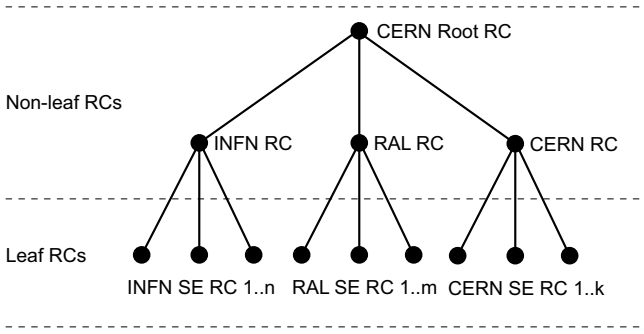

Figure 2.6: Outline of hierarchical, distributed replica catalogue

We assume that the protocol to talk to the Replica Catalog is HTTP. In the following example we describe some Replica Catalog lookup interactions using a redirection protocol based on HTTP. In order to indicate a redirection step in the RC, the Replica Catalog returns rcr://hostname where rcr stands for Replica Catalog Redirection and the hostname is the name of the host where the redirected request should be sent. In our HTTP example, the HTTP is used again to talk to the ”next” host.

If the client wants to look up a file, it has three possibilities:

- Connect to an Storage Element-Replica Catalog with an LFN directly. If the Storage Element has a physical instance of the file, its file handle is returned. If not, the lookup is not successful.
  In: http://SERC.cern.ch/getPhysicalFileNames?LFN=cms.org/triggers/myfile.db
  Out:pfn://ftp01.cern.ch/cms/triggers/myfile.db
- Connect to the Tier 1 Replica Catalog with the LFN. If it finds the LFN, it will return with a redirection request so that the client knows that it has to reconnect with the same query to a given Storage Element-Replica Catalog or that it can choose from a list of Storage Element - Replica Catalogs. If OUT: contains rcr://, then the client knows explicitly that redirection using an additional lookup step is required. (We may want to discuss alternative redirection mechanisms). If the LFN is not found, the lookup is not successful.
  In: http://SiteRC.cern.ch/getPhysicalFileNames?LFN=cms.org/triggers/myfile.db
  Out: rcr://SERC.cern.ch
- Connect to the CERN Root Replica Catalog. If the LFN is found, we get back a list of Tier 1 Replica Catalog sites that do have the given file. They can be contacted with the same query again to get a list of matching Storage Element - Replica Catalogs. Those then can be contacted for a real file handle. If the LFN is not found, it’s not in the system.
  In: http://RootRC.cern.ch/getPhysicalFileNames?LFN=cms.org/triggers/myfile.db
  Out: rcr://SiteRC.cern.ch

An API call like getPhysicalFilenamesFollowingRedirection(LogicalFilename) (see Figure 2.7) needs internal redirection steps if the lookup request is sent to a non-leaf Replica Catalog. If a request is sent directly to the Storage Element-Replica Catalog, no additional lookup step is required and the physical file location can be returned immediately. The redirection step is transparently taken at the client side rather than at the server side since the server will be the bottleneck in the system.

Each Replica Catalog needs to store the URLs of its parent replica catalogues. This information is required for loosely coupled batched update notification whenever entries are added/deleted to/from a leaf replica catalogue. Update notification propagates up the chain in the Replica Catalog hierarchy. The Grid Service Index is used to maintain parent-child relationships among Replica Catalogs.

The advantages of the hierarchical replica catalogue approach can be summarized as follows

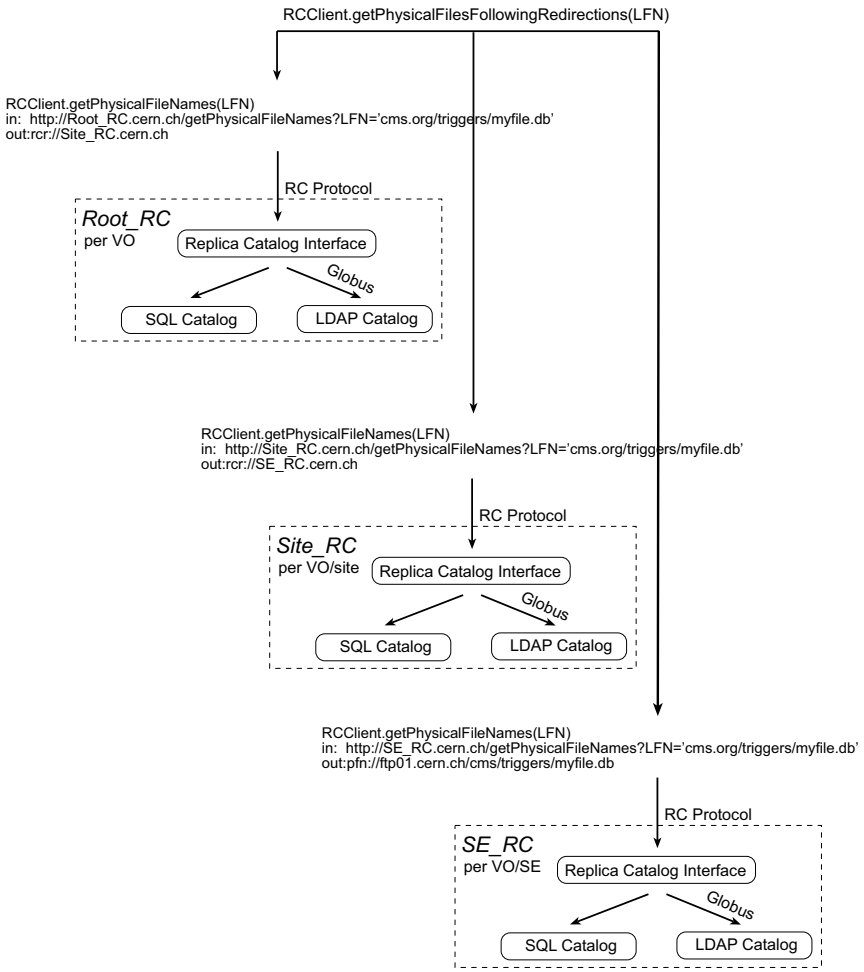

Figure 2.7: Interaction in hierarchical, distributed replica catalogue

- Scalable to a large number of sites and Storage Elements
- Each site or Storage Element is autonomous and can manage files locally

2.11.2 API

It will be the Replica Manager that uses the Replica Catalogue API. However, experienced users or other Grid tools like the Grid Scheduler might also have direct access to the Replica Catalogue. Both logical and physical files can carry additional meta data in the form of ”attributes”. Logical file attributes may include items such as filesize, CRC check sum and file creation timestamp. Physical file attributes may include a ”master” flag as well as catalog insertion and update timestamps. Additional standard attributes may be defined later.

- add/deleteLogicalFileName(LogicalFileName)
  Add a logical file in the replica catalogue.
- add/deletePhysicalFileName(LogicalFileName, PhysicalFileName)
- getPhysicalFileNames(LogicalFileName) Gets names of all physical files belonging to a logical file.
- add/deleteLogicalFileAttribute (LogicalFilename, AttributeName, AttributeValue)
- getLogicalFileAttributes (LogicalFileName): List of attribute/value pairs
- add/deletePhysicalFileAttribute (PhysicalFileName, AttributeName, AttributeValue)
- getPhysicalFileAttributes (PhysicalFileName): List of attribute/value pairs

2.11.3 Protocols

XML or URL encoding over https.

2.11.4 Services needed

Globus Replica Catalog (based on LDAP) and SQLDatabaseService.

2.11.5 Constraints and assumptions

None.

2.11.6 Open issues

We are interested in discussing what precisely a collection is, whether collectins are useful and how they could be used.

2.12 ReplicaManager

2.12.1 Introduction

The ReplicaManager has knowledge about file replicas (through the ReplicaCatalog service) and is responsible for consistent replica creation, moving and deletion. Note that maintaining consistency among replicas in different sites and storage locations is a task for the Consistency Service on top of the Replica Manager. The Consistency Service deals with updates of replicas and guarantees certain levels of consistency. Note that since most of the data is read-only, the consistency service is only applicable to data that is modified after it has been made available to the Grid. Research is necessary in order to define a detailed architecture.

The ReplicaManager is responsible for computing the cost estimation for replica creation. Information for cost estimates, such as network bandwidth, staging times and SE load indicators, are gathered from the IMS.

Policy issues are also handled by the ReplicaManager. For instance, before the RM can create a new replica at a given site it must check the local policy of that site. For example, does this person/group have write access? Is their quota full?

The RM is also responsible for keeping the RCs data synchronized with the actual files on the SE. Normally the SE should inform the RM when it deletes a file. However it is possible that the file be removed without the RMs knowledge. To handle this problem, the RM will periodically ask the SE if the replica is still there. More discussion is necessary. Maybe it is better to remove the entry if the pin request fails.

2.12.2 API

The API includes the following functions:

- addPhysicalFileName(LogicalFileName, PhysicalFileName)

  Enters the physical file as a replica of the logical file in the ReplicaCatalog, and assigns appropriate file attributes. If the logical file name does not yet exist, the physical file is marked as ”master”, otherwise as ”replica”. This method does not include any data movement. In other words, this method delegates to ReplicaCatalog.addPhysicalFileName(...) method, followed by ReplicaCatalog.setFileAttributes(...)

- deletePhysicalFileName(LogicalFileName, PhysicalFileName)

  Removes the association from the given LFN to the given PFN from the ReplicaCatalog and tells the SE holding the physical file that it is no more needed.

- getPhysicalFileNames(LogicalFileName)

  Returns the list of all known replicas of the logical file name. In other words, simply delegates to ReplicaCatalog.getPhysicalFileNames(..), possibly automatically following redirections.

- copy(PhysicalFileName source, PhysicalFileName destination, String protocol):Status

  This method select an appropriate file transport protocol, unless explicitly specified. Next, it determines the necessary transport file names for the selected protocol. Finally, it then delegates to the FileCopier.copyFile(TransportFileName source, TransportFileName destination) method, in order to copy a file from one StorageElement to another. Note that destination can also includes non SE locations such as localhost.

- copyAndAddPhysicalFile(PhysicalFileName source, PhysicalFileName destination, LogicalFileName lfn, String protocol):Status

  Chains together ReplicaManager.copy(...) and ReplicaManager.addPhysicalFileName(..) as one reliable atomic transaction. In other words, a file is copied from source to destination, and only if this is done successfully, the destination file is entered into the ReplicaCatalog as being a replica of the logical file.

- generatePhysicalFileName(LogicalFileName filename, PhysicalFileNamePattern)

  Generates a PhysicalFileName satisfying PhysicalFileNamePattern. Selects a StorageElement and delegates the name generation to it. PhysicalFileNamePattern specifies restrictions on the legal PFNs to be returned. An empty pattern indicates that the user is happy with any PFN. A pattern can specify a list of suitable SEs and desired directory prefixes to be prepended to generated names. Additional input parameters like filesize and access rights may be added later via a call to set attributes.

- estimateCostForCopy(PhysicalFileName source, PhysicalFileName destination, String protocol): Time

  This method select an appropriate file transfer protocol, unless explicitly specified. It then estimates the time that copy(..) would take.

- getLocationOfBestReplica (LogicalFileName) : PhysicalFileName

  based on cost functions (to be defined), the PhysicalFileName of the best replica is returned.

2.12.3 Protocols

To be decided. Potential candidates: LDAP, Soap/https, cgi/servlet over https, and many more

2.12.4 Services needed

- ReplicaCatalog, for storage and retrieval of metadata.
- StorageElement: for physical transport and disk pool management
- IMS: for monitoring information lookup
- Accounting for: Smart decisions in replica placement and selection

2.12.5 Constraints and assumptions

None.

2.12.6 Open Issues

Note that in this API we do not add the notion of collections. We are interested in user requirements related to file collections. Note that in response to user requirements a LFN is defined very liberal: It consists of a lfn:///¡hostname¿, followed by a "/" separator, followed by any arbitrary application specific string. Because arbitrary strings do not necessarily follow directory path conventions, a ReplicaCatalog does not and cannot associate hierarchical tree semantics with a LFN. Hence a file collection cannot implicitly be modelled as the set of files contained within a LFN "directory". As can be seen, the freedom of allowing arbitrary strings does have drawbacks, as it prohibits interpretation as directory paths. If applications require LFNs with directory path structure, then the LFN conventions may need to be revisited.

2.13 SoftwareRepositoryService

2.13.1 Introduction

The applications need to be available within the CEs so that they can be run. Typically applications make use of a a range of libraries some of which are the responsibility of the application and some which the application expects to find in place (e.g. glibc).

Dependencies exist between the different components (e.g. libraries) which make up an application.

If the pre-installed software environment within the CE to run a job is incomplete, we need a service which will, respecting dependencies, dynamically set up the environment and the application so that it can run.

To do this efficiently will require some kind of caching mechanism so we expect to make use of the data management services. However caches of old programs must be eliminated promptly.

2.13.2 Class and Object Diagrams

2.13.3 API

To be defined.

2.13.4 Protocols

To be defined.

2.13.5 Services needed

- ReplicaManager

2.13.6 Constraints and assumptions

Not yet considered.

2.13.7 Open issues

Not yet considered.

Chapter 3

Infrastructure

3.1 Introduction

The facilities listed here may at some stage become services.

3.2 Network

In the future, the network will also be a GridResource. Future network capabilities will include the ability to ask for various Quality of Service (QoS) levels, and the ability to make advanced reservations. For example, a grid user wishing to run a large interactive job might request simultaneous advanced reservations for compute element A and storage element B, and a premium QoS channel between those sites from 14:00 to 16:00 tomorrow.

These advanced network capabilities will be incorporated into the DataGrid project, depending on when they become available.

3.3 Fabric Management

3.3.1 Introduction

The functionality provided by a Computing Fabric can be classified into two main categories: User job execution on batch and interactive CPU services and administration of the computing fabric.

The ComputingElement (CE) has been described in section 2.2. It represents the interface to the fabric for the Grid. The administrative services presented here are *not* Grid services - they are intended to be used by system administrators and operators for running the systems maintenance.

The goal is the automatized deployment and management with reduced systems administration and operation costs of large computing fabrics containing O(100) to O(10K) nodes.

The main management functionalities are:

- *Node Installation*: Automated node installation and maintenance
- *Software Distribution*: Installation and management of application software packages on the nodes
- *Fabric Monitoring*: Job, node and service based monitoring inside a fabric
- *Resource Management*: User jobs and administrative tasks scheduling
- *Fault Tolerance*: Fault recovery at the node and fabric level
- *Configuration Management*: A central Configuration Database for all fabric configuration information

The fabric configuration information is stored in a central fabric configuration database. This information is used for (re)installing computer nodes according to defined profiles and deploying the requested software packages on them. A Resource Management System will efficiently schedule user jobs to the nodes and coordinate them with systems maintenance tasks. With the Fabric Monitoring service, node status information will be collected and correlated into service status and alarm views. The Fabric Monitoring will also allow for user job monitoring. Information from the Fabric Monitoring will be used by the Fault Tolerance system, which will be triggered on system misbehaviour for automated error recovery.

3.3.2 Functionality

Node Installation and Software Distribution

A bootstrap installation server will be responsible for the initial machine install. It will provide a node with an initial downsized system environment when (re)installing it. System parameters and site policies are applied to it according to the node's configuration stored in the Configuration Database.

A Node Management System agent running on each node will be responsible for fetching, installing, configuring, upgrading and verifying the software packages (system components and applications) which have been defined in the Configuration Database for this specific node or node profile (ie. disk server, farm CPU node, ...).

Configuration Management

A central Configuration Database will store alll hardware, system, application etc. configuration of a fabric. A graphical configuration editor, together with a command line interface, will be provided for accessing, querying and modifying the database's content. Modification requests are subject to validation. Access to the database will be securized (authentication/authorization).

Resource Management

The Resource Management System (RMS) coordinates user jobs with the execution of administrative tasks on fabric farm nodes. It will make sure that the integrity of user jobs is preserved and not affected by updates done to the system or the application environment. The RMS has always the control over the tasks running on a node. The RMS will be able to include and exclude nodes from the farms if required by an operator or the Fault Tolerance system. The RMS does also handle manage the Grid user jobs running on the fabric (see section 2.2).

Fabric Monitoring

The goal of the Fabric Monitoring is twofold. On one hand, the monitoring of performance, functional and environmental changes for all resources contained in a fabric in order to allow for optimised utilization of those resources. On the other hand, the monitoring system will provide an interface for applications to insert monitoring measurements.

The monitoring is used by fabric components to detect operational problems and trigger (automatic) remedy actions by the Fault Tolerance system. It is also used by the fabric managers and operators to get a health and status view of services and resources as well as accounting and history data.

Fault Tolerance

Faults on a node will trigger the execution of recovery actions defined by the Fault Tolerance system. Problems will be tried to be solved automatically on the node; if the problem can't be fixed locally (eg. a hardware failure, or CPU overheating), this will initiate a corrective action at the service level (eg. by moving the service to another node).

3.4 Security

3.4.1 Introduction

The security architecture supports the definition and management of credentials and security policies across the multiple security domains contained in the DataGrid project. Cryptographically secure communication mechanisms are an essential part of the Connectivity layer of the Grid architecture as defined by Foster, Kesselman and Tuecke [7].

While there are many requirements for Global, or Grid-wide, security credentials and policies, the architecture leaves the local site manager in full control of all local resources and policies. The global security credentials are mapped on to local credentials. Multiple local security implementations will be supported, including UNIX and Kerberos based solutions.

Wherever possible, security solutions will be based on existing standards to enable interoperation with other Grid projects. These standards, developed by the GGF and/or the IETF, include the Generic Security Services API (GSS-API) [12], the Grid Security Infrastructure (GSI) [6] from Globus, the Transport Layer Security (TLS) [11] protocols and X.509v3 [15] format identity certificates.

3.4.2 Security Services

One useful classification of general Network Security Services is the following:

- Authentication
- Authorisation
- Integrity
- Confidentiality
- Nonrepudiation
- Availability

These general network security services are also required in Grid computing. A subsequent version of the DataGrid Architecture will include a definition of the security Grid Services. In this version the general network services are described here with some comments on their use in the Grid.

The authentication service allows users, processes and machines to prove their identities. In the context of Grid computing this includes the support of single sign-on, whereby Grid users can authenticate once and then gain access to many grid resources without further authentication. Single sign-on is achieved by the delegation of user credentials to other processes such that the new process can act as if it were the user with some or all of the users rights. The user should be able to restrict the delegation such that the process will only acquire a limited sub-set of the users rights. A process possessing delegated credentials should in turn be able to delegate the restricted credentials on to yet another process.

The authorisation service allows the managers of resources to grant or deny access to these resources to authenticated users and processes. At the Grid Connectivity layer this is based on individual users or entities. The Collective layer will allow authorisation within a VO to be based on membership of groups or the ability to act according to a particular role or with a particular privilege.

The integrity service ensures that messages are received as sent without loss, addition or modification of the data transmitted.

Confidentiality protects the transmission of data from eavesdropping.

Nonrepudiation prevents either sender or receiver from denying the transmission or receipt of a message.

Availability. A number of security attacks, commonly called “Denial of Service” attacks, can result in the loss of availability or in a degraded performance.

3.4.3 Other security requirements

Site and security managers need to be able to record and log all security operations. The security audit service includes the logging of the initial sign-on to the Grid, all instances of mapping of global to local security entities, all instances of delegation of users rights together with the normal security auditing in the local security domain, be it a UNIX or a Kerberos based system.

Resource allocation, or quotas, together with the related accounting service, while using the global security entities is not directly part of the security architecture. These will be covered elsewhere.

3.4.4 Compute Element - Storage Element Security Issues

There are a number of important issues that must be resolved in order to provide file level authentication access between the CE and a local SE. Depending on what protocal is used (see Figure 3.1), there are different security models that are possible.

One model is to assume that CE uses local file semantics to open files. For example, if the CE is attached by a NFS or a SAN, then the standard Unix open commands will work, and security will be UID based. This has many security implications. The CE and SE must have the same user and/or group IDs, and the user and group read / write permissions must be set appropriately. For example, if a set of files is group read / writeable, how can we prevent one group member from overwriting another group members files? Supporting Grid supplied ACLs for file access would be very difficult under this model.

The second model is to assume that local file semantics are not available, and that all file access is via GridFTP. The GridFTP library supports most Unix IO-like file access (e.g.: open, close, read, write, seek), but not all (e.g.: lock). Using this method, all file access permissions can be checked via standard Grid authentication mechanisms. The problem with this method is that it requires all applications to replace their ”open” calls with ”gridOpen”. This may work for some applications, but with systems like Objectivity and Oracle, this will not be possible.

Most likely some hybrid of these two models will be required. We need to better understand the WP8-10 requirements and use cases before we can determine the optimal solution for this issue.

In addition to the file level security issues, there is also the issue of whether or not individual farm nodes can access remote SEs (or other remote Grid Services) directly. Many farms are configured so that only 1 node can access the internet, and all other nodes are on an internal subnet. In this case, there may need to be a gateway node to proxy Grid communication to non-local Grid Services, as shown in Figure 3.1. This issue is still under investigation.

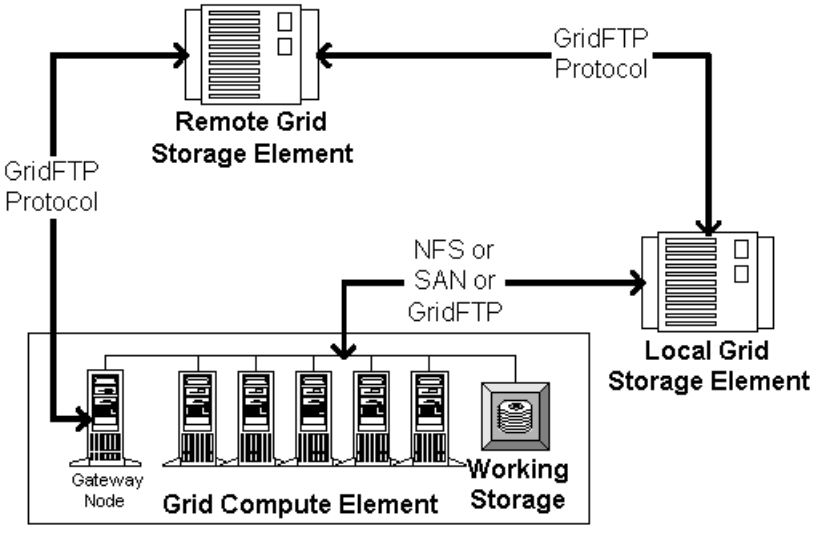

Figure 3.1: CE to SE connectivity and protocols

3.4.5 Open Issues

The task of specifying the security architecture has only just been started, so the number of open issues is in fact very large. Two of the many issues are currently described here.

Early implementations of the security authorisation service will support authorisation based on individual users and simple groups of users. A more complete VO “community authorisation” system, allowing more complex group structures, roles and privileges will be developed later.

As described above, the current authorisation scheme relies on the creation of local access control lists under the control of the local security implementation, UNIX or Kerberos. In the longer term there is likely to be a requirement for the definition of some form of global access control, where security entities can be described globally.

Chapter 4

Use Cases

To illustrate the use of the services described in this document, we describe a few simple Use Cases. These are not meant to demonstrate the complete functionality, but to point out the key capabilities of the architecture.

We describe each Use Case in words and show the collaboration or sequence diagram to show how this might be implemented with the services described in 2

Note: At the moment we only have one - and that has not been completely worked through yet

4.1 Submit and run a simple batch job to process one input file to produce one output file

This is almost the simples batch job one could imagine. The user specifies his job via a JDL file:

```
Executable=/usr/local/atlas.sh
Requirements = TS >= 1GB
Input.LFN = lfn://atlas.hep/foo.in
argv1 = PFN(Input.LFN)
Output.LFN= lfn://atlas.hep/foo.out
Output.SE = datastore.rl.ac.uk argv2 = PFN(Output.LFN)
```

and where the script to be submitted is:

```
#!/bin/sh
gridcp $1 $HOME/tmp1
grep higgs $HOME/tmp1 > $HOME/tmp2
gridcp $HOME/tmp2 $2
```

Figure 4.1 shows how the processing might be achieved. The user submits the job to the GS. The GS asks RM for list of all PFNs for the specified input file The scheduler then determines if it is possible to run the job at a CE that is “local” to one of the PFNs. If not, it locates the best CE for the job, and creates a new replica of the input file on a SE local to that CE. The GS then allocates space for the output file on the same SE, and pins the input file so that is not deleted or staged to tape until after the job has completed. Then the job is submitted to the SEs job queue. When the GS is notified that the job has completed, it tells the RM to create a copy of the output file at the site specified in the JDL. The RM then will tag this copy of the output file the “master”, and make the original file a “replica”.

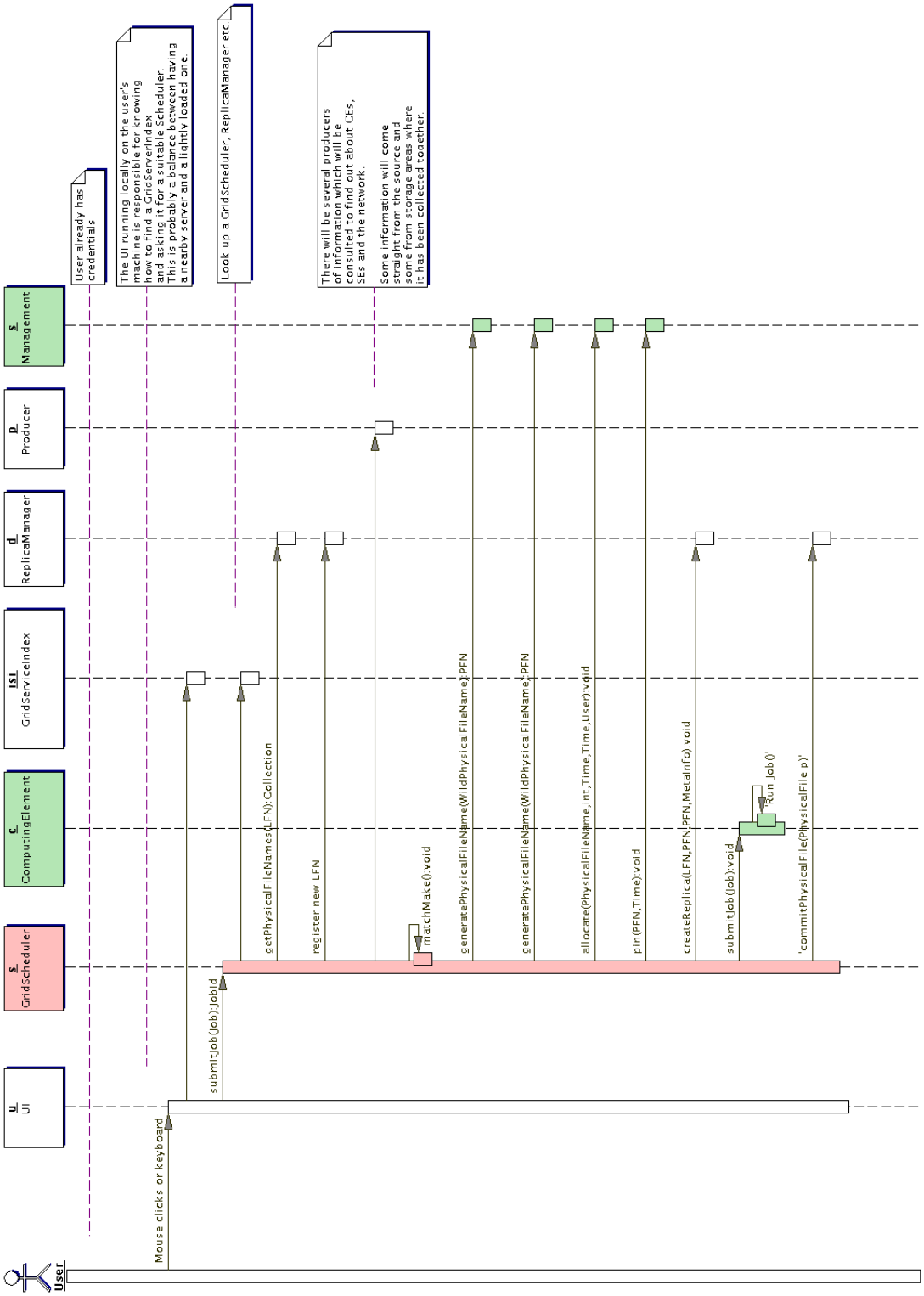

Figure 4.1: Run simple grep job

Chapter 5

Glossary of Acronyms

For more detail, please see the index.

CE ComputingElement; a Grid-enabled computing resource

GS GridScheduler; service responsible for selecting which grid resources to use for a given job.

GGF Global Grid Forum; http://www.gridforum.org/

GIS Grid Information Service; (e.g. IMS or Globus MDS)

GMA Grid Monitoring Architecture; monitoring architecture defined by GGF

GSI Grid Security Infrastructure (Globus Security mechanism)

IMS Information and Monitoring System; catalogue and distribute static and dynamic data about the Grid

JDL Job Description Language; to describe Grid jobs

LFN Logical File Name; A globally unique name to identify a specific file which is mapped by the RC onto one or more PFNs

LRMS Local Resource Management System; controls resources within a CE e.g. PBS or LSF

PFN Physical File Name; URL of actual physical instance of an LFN

RC ReplicaCatalog; associates a LFN to one or more PFNs

Replica A copy of a file that is managed by the Grid middleware

RM ReplicaManager; provides several services related to replicas

SAN Storage Area Network

SE StorageElement; a Grid-enabled storage system

TFN Transport File Name; URL to access a given file on a SE.

UML Unified Modeling Language; a notation for describing software systems

VO Virtual Organization; A set of individuals defined by certain sharing rules - e.g. members of a collaboration.

Acknowledgements

It is a pleasure to thank Ian Foster and Carl Kesselman who both acted as consultants to the ATF and gave us a lot to think about at the beginning of the project. We also wish to acknowledge the input offered by Ingo Augustin and Federico Carminati.

Appendix A

Classes

Description of Classes identified so far. This can be automatically generated from Together. *This will be provided later*

Appendix B

Responsibilities

Here we list our understanding of the responsibilities to realise the various components.

Component	WP	Comment
ComputingElement	1 and 4	
Consumer	3	
Fabric Management	4	
FileCopier	2	
GridScheduler	1	
Logging and Bookkeeping	1	
Network	7	
Producer	3	
ReplicaCatalog	2	
ReplicaManager	2	
Security	“Security”	
ServiceIndex	2	
SoftwareRepository	-	
SQLDataBase	2	
StorageElement	5	

Bibliography

[1] Grady Booch, James Rumbaugh, and Ivar Jacobsen. *The Unified Modeling Language Use Guide*. Addison-Wesley, 1999.

[2] CASTOR project status. http://wwwinfo.cern.ch/pdp/castor/presentations/chep2000/.

[3] Ewa Deelman, Ian Foster, Carl Kesselman, and Miron Livny. Griphyn data grid reference architecture. Draft, Jan 2001.

[4] ENSTORE data storage system. http://www-hppc.fnal.gov/enstore/.

[5] Steve Fisher. Relational model for information and monitoring. Technical report, GGF, 2001.

[6] I Foster, C Kesselman, G Tsudik, and S Tuecke. A security architecture for computational grids. 1998.

[7] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. Technical report, GGF, 2001.

[8] Basics of the high performance storage system. http://www.sdsc.edu/projects/HPSS/.

[9] M.Fowler and K.Scott. *UML Distilled*. Addison-Wesley, second edition, 2000.

[10] Practical uml. http://a1612.g.akamai.net/f/600/1325/9d/164.109.49.29/files/services/umlshortcourse/index.html.

[11] IETF, RFC 2246. ftp://ftp.isi.edu/in-notes/rfc2246.txt.

[12] IETF, RFC 2743. ftp://ftp.isi.edu/in-notes/rfc2743.txt.

[13] SATSTORE interface document. http://tempest.esrin.esa.it/ datagrid/docs/docs/SatStoreICD401.doc.

[14] Brian Tierney, Rich Wolski, Ruth Aydt, and Valerie Taylor. A grid monitoring service architecture. Technical report, GGF, 2001.

[15] ITU-T recommendation X.509.

Index