DataGrid

DATA MANAGEMENT (WP2) ARCHITECTURE REPORT

DESIGN, REQUIREMENTS AND EVALUATION CRITERIA

WP02: Grid Data Management

Document identifier:	**DataGrid-02-D2.2-0103-1_2**
Date:	**13/09/2001**
Work package:	**WP02: Grid Data Management**
Partner(s):	**CERN, IRST, SRC, UH, INFN, PPARC**
Lead Partner:	**CERN**
Document status	**DRAFT**
Deliverable identifier:	**DataGrid-D2.2**

Abstract: This is the architecture and design document of WP2. We present the motivation and the detailed description of all the services provided by WP2.

Delivery Slip

	Name	Partner	Date	Signature
From	Ben Segal	CERN	13/09/2001	
Verified by	Mark Parsons		13/09/2001	
Approved by				

Document Log

Issue	Date	Comment	Author
0_2	17/07/2001	First draft	Wolfgang Hoschek, Javier Jaen-Martinez, Peter Kunszt, Ben Segal, Heinz Stockinger, Kurt Stockinger, Brian Tierney
1_0	04/08/2001	Final draft	Same
1_1	24/08/2001	Draft for PTB of 29/08/2001	Same
1_2	13/09/2001	Final draft for PMB	Same

Document Change Record

Issue	Item	Reason for Change
1_0	Document structure	
1_1	Document structure & details	Input of WP1, WP5, WP8 reviewers
1_2	Document Introduction	Input of PTB review

Files

Software Products	User files
Word	DataGrid-02-D2.2-0103-1_2

CONTENT

1. INTRODUCTION

This document is produced by the Grid Data Management Work Package (WP2) of the European DataGrid Project. It gives a detailed overview of the architecture and design issues being addressed by WP2 in the course of its work.

This Work Package is developing middleware and documentation in a rapidly changing area of technology. Its output will evolve considerably over the three year period covered by the Project. The present document is based on work carried out over the first 6 months of the Project. Although complete in itself, and constituting a finished Project Deliverable, important additions to the material presented here will be published as Addenda in due course.

The Work Package deals with some topics of a clearly delimited nature (e.g. optimised data transfer techniques) but some other topics have a more diffuse or broader scope. Examples of the latter deal with Query Optimisation or Security issues: these involve interaction with other Work Packages or, as in the case of Security, involve Project wide considerations. As a result there is an evident unevenness of treatment imposed on us and sometimes this can affect the clarity of the discussion. For this, we apologise in advance.

Some parts of this document directly reflect WP2's contribution to the Architecture Task Force (ATF) document [R2]. The ATF is the overall architectural authority of the Project, and its documents must be consulted for any discussion of the broader context of WP2 within the DataGrid Project.

Nevertheless, we will attempt to give a short summary of WP2's main architectural emphases in the following section.

1.1. APPLICATION AREA

Data Management is a very broad concept. In this document we will discuss the architectural issues it presents in the EU DataGrid environment. We will define the scope of the Data Management Work Package (WP2), the design of data management services proposed for the DataGrid and the API's to the middleware we will deliver. This will be done in Chapters 3-6, based on a Use Case analysis presented in Chapter 2.

Before proceeding, we will give some background information to help the reader to understand our priorities and main areas of emphasis.

1.1.1. WP2 Tasks

Because of the breadth of the issues involved, and the limited resources present in WP2 to address them, a process of choice was applied and a set of **Tasks** extracted. The resource management of the Work Package is partly based on these Tasks, and progress in the WP2 development effort depends to a considerable extent upon the staffing and individual success of the WP2 Tasks. Each Task was allocated approximately equal initial staffing resources.

The basic **Task Areas** of WP2 are:

- Data Access & Migration
- Data Replication
- Meta Data Management
- Secure & Transparent Data Access
- Query Optimisation

The character and difficulty of the above Tasks vary considerably and this determines the nature and degree of progress to be expected in the respective areas. Some Task areas have a larger existing body

of work than others. Furthermore, there are dependencies between certain Tasks from an architectural point of view. Simply put, once all this is taken into account, the area of Data Access & Migration is of lowest difficulty for WP2; the areas of Data Replication, Meta Data Management, and Secure & Transparent Data Access are of medium difficulty; and that of Query Optimisation is of greatest difficulty.

We therefore expect to make the most concrete progress in the areas of lowest and medium difficulty. We note that Query Optimisation is the most "research oriented" and involves the most inter- Work Package dependence, particularly with Work Package 1 where decisions on, for example, Job Scheduling and job decomposition must be coordinated with those of data replication and availability.

1.1.2. Areas of Innovation

The principal areas within which WP2 has already made important innovations or advances are:

- Distributed Hierarchical Replica Catalog Design
- SQL DataBase Service and Service Index Service for Meta Data Management
- Security Models for Data Access
- Query Optimisation

These areas are consequently treated in greater detail in the document than are some other areas.

1.2. DOCUMENT STRUCTURE

As a general rule, we have tried to improve readability by placing the more detailed material towards the end of this document.

Chapter 1 (the present chapter) is an introduction, placing the work of WP2 in context with the rest of the EU DataGrid Project's middleware developments.

Chapter 2 describes **use cases** that have driven the architecture and design of the Data Management Work Package.

Chapter 3 deals with the **architecture** of the DataGrid, as described in the ATF document [R2] with a special focus on Data Management; here we define the necessary terms to put Grid Data Management into context.

Chapter 4 discusses **general design issues** for WP2 and Chapter 5 deals with **more detailed design** of the identified components.

Chapter 6 gives a detailed description of the **services** WP2 will provide, including their **API's**. (For the moment these API's are shown schematically and a few are still undefined).

1.2.1. Planned Addenda

Although not an exhaustive list, planned future addenda to this document will cover:

- More details of the WP2 **API's**, including for example their error return codes. In the case of missing API's referred to above, these will be completed.
- Detailed **interactions** of WP2 components, which are being developed in terms of UML diagrams.
- More details in the area of **Query Optimisation**, including that of Simulation.
- More work on error recovery and robustness.

1.3. APPLICABLE DOCUMENTS AND REFERENCE DOCUMENTS

Reference documents

[R1] Report of the Steering Group of the LHC Computing Review, S. Bethke, H.F. Hoffmann et.al, http://cern.ch/lhcb-comp/Reviews/LHCComputing2000/Report_final.pdf

[R2] German Cancio, Steve M Fisher, Tim Folkes, Francesco Giacomini, Wolfgang Hoschek, and Brian L Tierney. The DataGrid Architecture. Version 2, July 2001. http://cern.ch/grid-atf/doc/architecture-2001-07-02.pdf

[R3] Paolo Busetta, Mark Carman, Michele Nori, Luciano Serafini, Floriano Zini, Using BDI Agents for a DataGrid Simulator, http://sra.itc.it/events/agents4Grid/program.html

[R4] CASTOR project status. http://wwwinfo.cern.ch/pdp/castor/presentations/chep2000/

[R5] ENSTORE data storage system. http://www-hppc.fnal.gov/enstore/.

[R6] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. Technical report, GGF, 2001.

[R7] Basics of the high performance storage system. http://www.sdsc.edu/projects/HPSS/

[R8] Asad Samar, Heinz Stockinger. Grid Data Management Pilot (GDMP): A Tool for Wide Area Replication, *IASTED International Conference on Applied Informatics (AI2001)*, Innsbruck, Austria, February 19-22, 2001.

[R9] SATSTORE interface document. http://tempest.esrin.esa.it/dataGrid/docs/docs/SatStoreICD401.doc

[R10] Heinz Stockinger, Asad Samar. GDMP User Guide. http://cmsdoc.cern.ch/cms/Grid , 2001.

[R11] Heinz Stockinger, Asad Samar, Bill Allcock, Ian Foster, Koen Holtman, Brian Tierney. File and Object Replication in Data Grids, *10th IEEE International Symposium on High Performance and Distributed Computing (HPDC2001)*, San Francisco, California, August 7-9, 2001.

[R12] Kurt Stockinger, Dirk Duellmann, Wolfgang Hoschek, Erich Schikuta. Improving the Performance of High Energy Physics Analysis through Bitmap Indices. 11th International Conference on Database and Expert Systems Applications, London - Greenwich, UK, Sept. 2000. Springer-Verlag.

[R13] Luciano Serafini, Heinz Stockinger, Kurt Stockinger, Floriano Zini, Agent-Based Query Optimisation in a Grid Environment, IASTED International Conference on Applied Informatics, Innsbruck, Austria, February 2001

[R14] Kurt Stockinger, Design and Implementation of Bitmap Indices for Scientific Data, International Database Engineering and Applications Symposium, Grenoble, France, July 2001, IEEE Computer Society Press

[R15] Workload Management Work Package, http://www.infn.it/workload-Grid/documents.htm

[R16] XSQL Documentation, http://www.cern.ch/javatree/share/opt/jdbc/xdk-9.0.1.0.0/xdk/doc/java/xsql/readme.html

[R17] Heinz Stockinger, Andrew Hanushevsky. HTTP Redirection for Replica Catalogue Lookups in Data Grids, http://www.cern.ch/hst/HTTP/http_redirection.ps, to be published.

[R18] Dirk Düllmann, Wolfgang Hoschek, Javier Jean-Martinez, Asad Samar, Ben Segal, Heinz

Stockinger, Kurt Stockinger. Models for Replica Synchronisation and Consistency in a Data Grid, *10thIEEE Symposium on High Performance and Distributed Computing (HPDC-10)* , San Francisco, California, August 7-9, 2001. See also: http://grid-data-management.web.cern.ch/grid-data-management/docs/replica_consistency.ps

[R19] Paul Avery, Julian Bunn, Takako Hickey, Koen Holtman, Miron Livny, Harvey Newman, CMS Data Grid System Overview and Requirements (DRAFT V9), http://kholtman.home.cern.ch/kholtman/

[R20] Kipp E.B. Hickman, *The SSL Protocol*, 1995, http://www.netscape.com/eng/security/SSL_2.html.

[R21] Goodwill J. Using Tomcat 4 security realms http://www.onjava.com/pub/a/onjava/2001/07/24/tomcat.html

[R22] J. Anderson, Information security in a multi-user computer environment, in *Advances in Computers*, vol. 12. New York: Academic Press, 1973, pp. 1-35. (I A1, SFR)

[R23] W. Stallings, Cryptography and network security: Principles and practice (2nd Edition), Prentice-Hall, 1999.

[R24] R.C.Summers, Secure Computing: Threats and Safeguards, McGraw-Hill,1997.

[R25] An Introduction to Role Based Access Control." NIST CSL URL: http://csrc.ncsl.nist.gov/nistbul/csl95-12.txt 11/1/00.

[R26] I. Foster, C. Kesselman, G. Tsudik, S. Tuecke, A Security Architecture for Computational Grids. *Proc. 5th ACM Conference on Computer and Communications Security Conference*, pg. 83-92, 1998.

[R27] B.W. Lampson, Protection, Procs. 5th Annual Conf. on Info. Sciences and Systs.,1971, 437-443. Reprinted in ACM Operating Systs. Review, 8, 1 (January 1974), 18-24.

[R28] Gollmann, D. *Computer Security*. John Wiley & Sons, New York, 1999

[R29] R.Sandhu et al., Role-Based Access Control models, Computer , vol. 29 , No2, February 1996, 38-47.

1.4. TERMINOLOGY

Definitions

Glossary

ACL	Access Control List; used to enable/disable kinds of access to resources.
AOD	Analysis Object Data; used in HEP: information used in final analysis.
CE	Computing Element; a Grid-enabled computing resource.
ESD	Event Summary Data; used in HEP: information required for detailed analysis and high-level reconstruction.
EDG	European Data Grid; (project name is often written “DataGrid”).
GDMP	Grid Data Mirroring Package; a WP2 application.
GS	Grid Scheduler; service responsible for selecting which Grid resources to use for a given job.

GGF	Global Grid Forum; see 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).
HDF5	Hierarchical Data Format by NCSA, version 5. Library and file format for storing scientific data.
HEP	High Energy Physics.
HSM	Hierarchical Storage Manager. HSM software allows infrequently accessed data to be migrated to less expensive offline storage automatically.
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	Replica Catalog; associates an LFN to one or more PFNs.
Replica	A copy of a file that is managed by the Grid middleware.
RM	Replica Manager; provides several services related to replicas.
SAN	Storage Area Network.
SE	Storage Element; a Grid-enabled storage system.
SRV	Short for Server.
TAG	Event Selection Tag; used in HEP: used for fast event selection.
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.

1.5. ACKNOWLEDGEMENTS

We would like to thank everyone in WP2 for their contributions. We had very useful discussions with members of the ATF and of other work packages, especially WP1 and WP4. The requirements from WP8-10 were also very valuable inputs to this document. Special thanks to our reviewers Frank Harris (WP8), Al Werbrouck (WP1) and Jules Wolfrat (WP5) for their careful and extensive comments.

2. DATA MANAGEMENT USE CASES

We define use cases in the broad sense of generic usage scenarios for the DataGrid as a whole, from which we deduce the requirements for data management. The two main categories are the generic Grid internal use cases and the application use cases.

2.1. GENERIC GRID USE CASES

This section deals with the data management use cases resulting from the generic Grid model. Our understanding of the detailed Grid architecture presented in the next chapter is based on this model.

2.1.1. Grid Computing and Storage Model

The DataGrid has two basic building blocks: StorageElements (SE) and ComputingElements (CE). A ComputingElement is the interface to computing power. The scheduler can run Grid jobs only on CE nodes. A Grid StorageElement (SE) is the generic name for any storage resource that includes a Grid interface. Both the SE and CE are part of the Grid "fabric", as defined in the *Anatomy of the Grid* paper [R6]. StorageElements may include large HSM systems like HPSS [R7], CASTOR [R4], SATSTORE[R9], and ENSTORE [R5], and will also include Grid managed disk pools. WP2 has to provide means to transport and access data between SE and CE nodes.

CE and SE may be either local or remote to each other with respect to network connection. The scheduler needs WP2 tools to be able to decide whether to replicate data necessary for job execution at an SE local to the chosen CE or to access the data at a remote SE. 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 scheduler 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.

2.1.2. Grid Monitoring and Scheduling

The DataGrid has of course more to it than just computing power and storage space. All the issues about scheduling and monitoring the system are being addressed as well. There are two other dedicated work packages for these tasks (WP1 and WP3). The ATF document [R2] describes in more detail how these components interoperate. At this point we just mention that most of the services provided by WP1 and WP3 need the storage of persistent Grid-Metadata which is also in the domain of WP2. Such metadata needs to be stored and will need to be accessible from throughout the Grid.

2.2. GRID APPLICATION USE CASES FOR HEP

We focus on High Energy Physics (HEP) use cases because of our more detailed understanding of them and based on documents provided by the HEP community (see for example [R1, R19]). We have worked closely with several experiments, particularly CMS.

Use cases from Biology and Earth Sciences will be incorporated in this document in a later version.

The following are three examples of activities that will be performed by the HEP experiments on the Data Grid:

- **Testing of algorithms for data production.** This activity will be performed by physicists in order to tune the algorithms that they want to use to produce higher-level data types from lower-level data types (e.g., AOD data from ESD data).
- **Data production.** This activity will be performed by "data production managers", in order to obtain a full conversion of data from a lower-level format to a higher-level format. Simulations of data production are also within this category.
- **Analysis.** This activity will be performed by physicists, in order to derive physics entities from the data.

Job Type	*User*	*Input Data Set*				*Output*
		Scheduled	*Known*	*Size*	*Spans Datatypes*	
Testing	Physicist	No	Yes	Small Subset	No	Data Subset
Production	Manager	Yes	Yes	All Events	No	Data Set
Analysis	Physicist	No	Partially	Small to All	Yes	Histogram/Tag

Table 2.1: The table outlines the similarities and differences between the three types of activities

Production and testing activities will be performed massively in the earlier phases of the HEP experiments, while analysis will become far more frequent and important over the lifetime of the experiments.

The analysis process differs from the other types of activities in certain important ways. Unlike the other activities, the output of the process is generally private data (such as a histogram or a collection of tags), which will not be managed by the Grid. The creation of summary statistical data like histograms also implies the need for an aggregation step not present in the other processes. (The aggregation step brings together the output data from the execution of algorithms on individual events). More interestingly, it is harder to know what data an analysis job will access, and in some cases the data accessed in a single job will come from more than just one datatype (e.g. AOD and ESD together).

It is clear that analysis is the most complex process in terms of data management. To investigate optimisation strategies for analysis is one of the most challenging tasks of WP2 and will be discussed in Chapters 4 and 5.

In the following, each of these use cases is discussed in more detail. We try to deduce the resulting requirements on Grid Data Management.

2.2.1. Algorithm Testing

Testing of algorithms and other application code in the Data Grid environment is one of the main activities in the initial phase of the project. This activity will never lose importance as new code is always being produced and needs testing. Tests can be simple or complicated, depending on what needs to be tested. In general it can be assumed that for most testing the input data are known in

advance, the run time will be short, and the output data is manageable (i.e. small). In terms of data management design, this is the least challenging activity: the input and output data need to be accessible to the test programs in a transparent manner. The application need not know anything about the physical location of the data on disk, the Grid should take care of this. Nevertheless, manual triggering of replication and local data access should be possible.

2.2.2. Data Production

This activity is present from the very beginning in form of detector simulations. The “data challenges” of the HEP experiments involve large scale simulations of the whole system, producing already a vast amount of data at an early stage. The data volume of the simulations increases in time so that it matches up with the real data volume by the time the detector comes online. The Grid Data Management architecture should scale up to the expected data volumes (see [R1].

2.2.3. HEP Analysis

Physicists will analyze the events generated in the LHC by iteratively selecting more and more restricted subsets, in order to isolate only the "interesting" events from the 10^9 events generated each year by an experiment. Selection is performed by defining cut predicates, that are applied in sequence to smaller and smaller sets of events.

A cut predicate p is applied to an input event set S and yields another set of events $S' = \{e \in S \,/\, p(e) = true\}$ as output. Cut predicates select those events which satisfy some properties defined on values of data products belonging to the events.

In the earlier phases of the analysis, a cut predicate p(e) is evaluated using the values of TAG data products. Data selection is specified by cut predicates defined as a set of constraints on certain TAG attributes of the events in the set (e.g., boolean expressions containing "greater-than", "less-than" and "equal-to" operations are used to restrict the input data set). During TAG data analysis, physicists typically "cut" the table data down to around 2% to 20% of its original size, until the selective power of this data type is exhausted.

In the later phases of analysis, physicists exploit values of more "low level" data products, such as AOD, ESD or even Raw data to perform further selection on the event set. In these phases, application of a cut predicate on an input data set is performed by submitting a *job* to the Data Grid. A job is a user-supplied piece of code and a specification for an input data set and an output data set. The piece of code executes the event selection based on the values in the input data set and produces the output data set possibly combining or aggregating data values of the selected events.

Cut predicates are created by stepwise refinement. In each step a new version of the predicate is applied to the input event set, then the output event set is examined to decide whether the current version of the predicate is useful or not to continue the analysis.

2.2.3.1. Analysis Scenarios

We will be considering the following scenarios.

1. A physicist is doing HEP analysis and he/she has been working on the highest level data (TAG data). These first phases of analysis may be performed outside the Data Grid. Having completed part of the analysis, the physicist then "runs out" of information at the TAG level and is forced to access lower level data (AOD) to continue the analysis. The physicist starts to submit jobs to the Grid. These jobs specify some AOD data products as input data set and define a cut predicate on

the input data set, in the form of a piece of code. The location for the output data set is also specified by the job.

2. The physicist has continued analysis on AOD data until she has exhausted information at that level. The rest is similar to scenario 1.

3. The physicist has continued analysis on ESD data until she has exhausted information from them. Again the rest is similar to scenario 1.

4. The fourth scenario does not follow from the previous one, but refers to a more general form of physics analysis, in which the physicist performs analysis on multiple datatypes concurrently. This type of analysis is possible because objects in each datatype may contain pointers to objects in other (lower level) datatypes, as well as to objects within the same datatype.

We are considering the case where a physicist user of the Data Grid is entering an analysis job to be performed on a particular data set (either AOD, ESD, Raw Data, or a combination of them). The output of the job could be considered to be TAG data, or some other data format which is transparent to the Grid. To submit the job the user provides the following information:

1. The input data set. This is specified basically as a set of logical filenames (LFNs).

2. A reference to a piece of code, which is to be executed on the input event set. The code contains an algorithm which is an atomic function for each event, and thus needs to be executed separately on the data for each event. (The code represents the computational job to be performed. Note that we assume this code can be transported for compilation and computation anywhere on the Grid.)

3. A physical location to deliver the output data to. This location may be specified automatically by the local application, and could lie "outside" the Grid, i.e. on the physicist's local PC or on a private server, which is not managed by the Grid. The output location may also be "inside" the Grid and added to the Replica Catalog to be processed further by followup jobs.

4. (Optional) processing hints, that help the Grid scheduler to find the best location where to run the job and help the Grid optimisation services in their activities.

Note that the input data product set of a job is actually a superset of the real data product set that will be accessed by the job.

2.2.3.2. Navigation

The input data set may not be completely known a priori (before the job starts execution) because the job might have to access some data products based on the computation it has performed, and of course this is only known at run-time. We refer to this as *navigation* between data products. In such cases (primarily in analysis type jobs) it is impossible to know precisely in advance what data the job will access, or how long it is likely to run for.

The Data Management architecture should allow for an efficient analysis activity and should handle the navigational aspect also. We have to provide the means to locate and access data available through the Grid at application run time. This motivates much of the functionality of the ReplicaManager and ReplicaCatalog services of our work package.

3. ARCHITECTURE

3.1. DATA MANAGEMENT VIEW OF THE GRID ARCHITECTURE

Figure 3.1 shows the layered Grid architecture, as discussed in [R6]. The components are the same as those defined in the ATF document [R2].

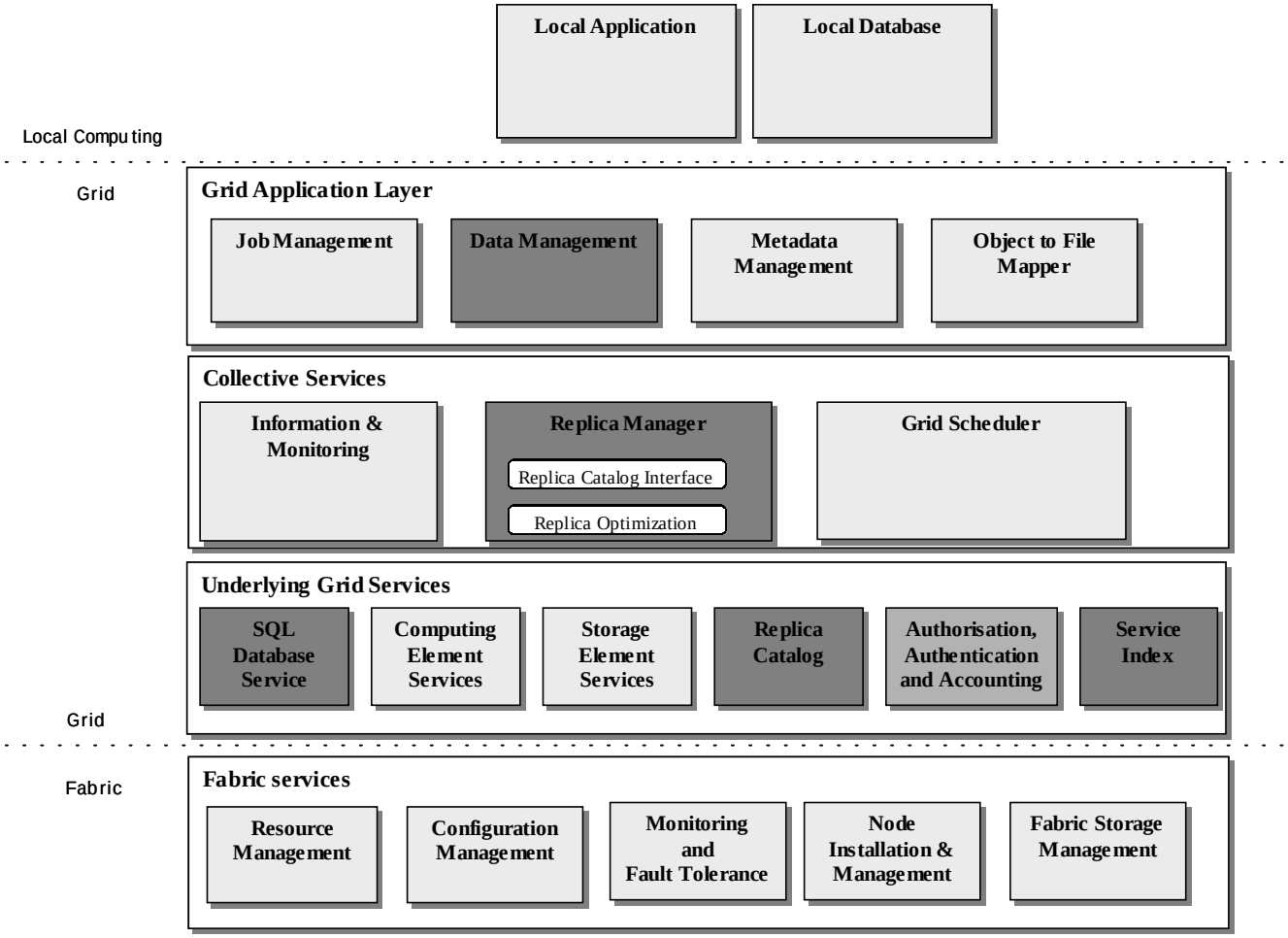

Figure 3.1: Grid Layered Architecture, WP2-centric view. The shaded items are completely in the domain of WP2 with the exception of Authorization, Authentication and Accounting, where only specific issues are handled (hence the lighter shading).

The Grid middleware that we are talking about in this document lives in the two lower Grid layers of Figure 3.1. It is nevertheless important to describe the other layers for the broader context of our services. The following is a description of the different services provided within each layer of the architecture. We have marked "WP2" those services provided by the Data Management Work Package.

3.2. LOCAL APPLICATIONS

The top layer of the architecture exists outside of the Grid infrastructure. The human users of the Grid will all be accessing the Grid through this layer, consisting of local desktop machines or servers. Data needs to be exported to this layer upon request, usually in the framework of data analysis where the users want to work with a small subset of the data available in the Grid locally. It is essential for interactive usage that this layer has efficient bindings to the Grid layers. The requirements for security are also driven by the heterogeneous environment of local users.

3.3. GRID APPLICATION LAYER

The applications which reside in the "Grid Application Layer" act as a means of interface and coordination between the local applications. The components we list below are just possible modules that a HEP experiment or any other Virtual Organization (VO) might want to provide. There may be less or more, depending on the needs of the VO. In our list we tried to incorporate those which are most likely to exist so that we can think about their interaction with our Grid middleware. Please be reminded that this is an evolving system: new application middleware requirements will need to be incorporated into the Data Management architecture if necessary.

3.3.1. Job Management

Each application will most likely have some sort of internal Job Management system, making decisions regarding which jobs are submitted to the Grid and at which time. Such decisions would be made based on application quotas, user priorities, expected run-times, and so on. (The idea here would be to stop "small time users" from unwittingly submitting very large jobs to the Grid, which drain its resources unnecessarily.) The Job Management system would also be responsible for scheduling large production type jobs. However, much of the responsibility could go directly to the Grid Scheduler.

3.3.2. Data Management (WP2)

Databases and data stores are used to store data persistently in a Data Grid. Several different data and file formats will exist and thus a heterogeneous Grid-middleware solution needs to be provided. Particular database implementations are both the choice and responsibility of individual virtual organizations, i.e. they will not be managed as part of Collective Grid Services.

Once a data store needs to be distributed and replicated, data replication features have to be considered. If a single, homogeneous data store existed which provided efficient and secure data replication over the wide-area network, such replication tools might be preferable. However, current storage technology is not sufficient (e.g. Objectivity's Data Replication Option is not optimal; ROOT does not provide replication at all; Oracle still needs to be evaluated), so high level replication tools have to be provided by the Grid middleware. Since a file is the lowest level of granularity dealt with by the Grid middleware, a file replication tool is required.

In general, successfully replicating a file from one storage location to another one consists of the following steps:

- *pre-processing*: This step is specific to the file format (and thus to the data store) and might even be skipped in certain cases. This step prepares the destination site for replication, for example by creating an Objectivity federation at the destination site or introducing new schema in a database management system so that the files that are to be replicated can be integrated easily into the existing database.

- *actual file transfer*: This has to be done in a secure and efficient fashion; fast file transfer mechanisms are required.
- *post-processing*. The post-processing step is again file type specific and might not be needed for all file types. In the case of Objectivity, one post-processing step is to attach a database file to a local federation and thus insert it into an internal file catalog.
- *insert the file entry into a replica catalog*: This step also includes the assignment of logical and physical filenames to a file (replica). This step makes the file (replica) visible to the Grid. (more details on replica catalogs, logical and physical filenames can be found in Chapter 4).

A generic file replication software tool called Grid Data Mirroring Package (GDMP) is provided by WP2 and implements all the above steps using several underlying Grid services. For further details see Chapter 4. It is important to point out that several different data stores can be supported but storage system dependent plug-ins have to be provided for the pre- and post-processing steps.

3.3.3. Object to File Mapping

The high level experiment's data view contains neither the concept of files nor the concept of data replication: all objects are supposed to simply "exist" without regard to how they are stored and how many replicas exist [R11]. Files and replication appear only at lower layers of abstraction as implementation mechanisms for the experiment's object view. A single file will generally contain many objects. This is necessary because the number of objects is so large (of the order of 10^7 to 10^{10} for a modern physics experiment). An object to file mapping step is required and needs to be provided by the persistency layer of the individual experiment. Thus, Grid middleware tools only deal with files and not with individual objects in a file.

3.3.4. Metadata Management

In addition to the actual data to be stored persistently (event data in case of High Energy Physics), experiment specific metadata about files might need to be stored. The metadata may contain information about logical file sets (e.g. a particular set of files contains certain physics objects etc.). In principle, several files might be part of several sets or collections. In our current understanding such VO-specific metadata is managed by the VO's software infrastructure and not by DataGrid middleware tools.

3.4. THE COLLECTIVE SERVICES LAYER

The Collective Layer represents the set of high level (but generic) Grid services offered to Grid enabled applications. The services provided in this layer can be roughly separated into three areas, information (monitoring) management, data management, and computation management, corresponding to Work Packages 3, 2 and 1 respectively. In Figure 3.1 we label these components according to the names of the major services these work packages will provide: Information and Monitoring, Replica Manager and Grid Scheduler.

We focus our discussion in this document on the data management issues. See the corresponding documents of WP1 and WP3 for more details on the Scheduler and Monitoring respectively.

3.4.1. Replica Manager (WP2)

Our detailed design for replication is presented in Chapter 5. At this point we just mention the two major components of the Replica Manager in the Collective Layer: the interface to the Replica Catalog and the interfaces to replica selection/optimization and other high-level replication services.

3.5. THE UNDERLYING GRID SERVICES LAYER

The Grid services on the next-to-bottom level of the diagram represent the basic and essential services required in a Grid environment. These services include the ability to schedule jobs on remote clusters, to transfer files efficiently between sites, to pass messages between processes, and to control authorisation and access to files and services. One can view these services as Grid versions of standard LAN services, such as those offered by the UNIX operating system.

Our work package will provide some of these services. (More detailed design of each of our services is discussed in the next two chapters).

3.5.1. Computing Element Service

Under this generic name we collect all services that enable communication with a Grid Computing Element. This component is responsible for submitting jobs to the Fabric Layer and interfacing them to the Grid Scheduler. WP1 and WP4 will provide these services.

3.5.2. Storage Element Service

This is the interfacing service to the Storage Element. We will provide a data transfer interface to the Storage Element (the FileCopier service).

3.5.3. Replica Catalog (WP2)

The Replica Catalog stores information about physical files on any given Storage Element. This is one of the services which will be provided by our work package.

3.5.4. SQLDatabase Service (WP2)

This is the service we plan to provide in order to store Grid Metadata. This service may be used by any other Grid middleware to store persistent metadata.

3.5.5. Service Index (WP2)

The Service Index stores information on Grid services using the SQLDatabase Service to store the metadata. This is also provided by WP2.

3.5.6. Security - Authorisation, Authentication and Accounting (WP2)

All the services concerning security are schematically indicated by the box “Authorisation, Authentication and Accounting”. We focus only on the aspect of data security, which is one of the tasks of WP2. The general security issues of the whole DataGrid Project are much wider than WP2 can address.

3.6. FABRIC SERVICES LAYER

Fabric Services are the low-level layer that the Grid middleware actually runs on. We refer the reader for further discussion of Fabric Services to the ATF document [R2] and to the detailed design documents of WP4.

4. DESIGN ISSUES

4.1. INTRODUCTION

In this chapter we describe the major design issues of WP2, leading to the detailed designs we present in the following chapter. This chapter deals with the following:

Data Granularity

Files are the data units we decide to deal with for now. (We understand that Grid Object management is on the wish list of some applications and we are investigating how to enable Grid Object management in our Optimisation task).

In the current WP2 context, data is managed in units of named files. In the future we will also introduce collections of files. In the short term, the mapping of files to collections is defined by the Grid applications, not by the Grid middleware.

Replication System

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 optimising I/O across the Grid, creating data replicas where appropriate.

The replication system includes all services necessary to provide consistent data replication across DataGrid nodes. The services provided are the Replica Catalog and Replica Manager. More services may evolve in the future, like Replica Selection and a Consistency Service.

Metadata Storage

We describe how we intend to handle persistent Grid Metadata storage to be used by WP2 as well as other Grid Services if they desire.

Grid metadata consists of data on the Grid internals, like Grid layout information, monitoring, past usage statistics, user information, etc. In order to manage the metadata gathered by the Grid, we have the SQLDatabase service that can be used as a persistent data store by any other Grid service. An example of a WP2 service using it to provide information on existing Grid services is the ServiceIndex service.

Security

In our limited context of file access control we have a design for security in mind. The full security issue of the Grid is beyond the context of WP2. We do investigate interfacing Grid security to an HSM and to an SQL data base service.

Optimisation

Optimising data access is an active research topic. We investigate future directions for Grid Data Management for an integrated optimised data query and access. We do talk about objects and files, discuss a possible job description language and give insights into query optimisation. The aim is to collaborate with application developers to enhance data access.

4.2. FILE REPLICATION

With respect to replication, 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 Storage Element 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 (e.g. for guaranteeing that when a master file is removed, all its replicas are also removed) are described further in Chapter 5. 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 also described in Chapter 5. Master files would typically be stored on a "reliable" system, (i.e. backed up), whereas a replica does not require backup.

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 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, whose API is given in Chapter 6, 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 files are removed, subject to their "lifetime" attributes.

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 the same file will be used 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 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 database 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.

4.2.1. File and Replica Identifiers

Since a physical file can have several identical instances (replicas), a naming convention is required that assigns a logical file name to a set of replicas (physical file names). A logical file name (LFN) is

used to uniquely identify a set of identical physical files at different storage locations (StorageElements). Each physical file has a unique physical file name (PFN).

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 (TFN) - 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. In this context we require that no two LFNs point to the same PFN, i.e. a physical file cannot have more than one logical name. This is a purely technical requirement that makes it easier for us to implement the basic functionality of the system; this can be relaxed at a later time if necessary.

Each virtual organization, 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.

4.2.2. Logical File Name (LFN)

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 **virtual hostname**, 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 such 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 LFN "hostname" does *not* tell where a file is physically stored, nor does it tell how the file can be accessed. The virtual hostname syntax facilitates easy and scalable world-wide cross-organisational name space partitioning. It is included to show that all namespaces below it belong to (and are managed by) the owner of the hostname. To assure uniqueness, it is recommended that the virtual hostname be also present in DNS.

4.2.3. Physical File Name (PFN)

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**, as 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

4.2.4. Transport File Name (TFN)

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 normally not use the TFN directly, as the translation from the PFN will be handled by the middleware. A TFN may only be meaningful locally. The TFN is more than just the PFN and a protocol since, depending on the protocol, complicated translation mechanisms may be involved.

A transport file name is a URL, with a set of supported protocols (e.g. Gridftp, ftp, http, rfio and file). URL encoding for commands and parameters within a transport file name is not allowed. As for the PFN case, the TFN hostname which follows the protocol specifier is that of the corresponding SE.

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
- rfio://wolfy.cern.ch/datastore/file.dat

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 their scheme prefix.

4.2.5. Replica Manager

The ReplicaManager (RM) has knowledge about file replicas (through the ReplicaCatalog service) and is responsible for consistent replica creation, moving and deletion. In addition, since all replicated files need to appear in a global name space and need to be uniquely identified, the ReplicaManager is responsible for inserting logical and physical file information into the ReplicaCatalog.

4.2.5.1. File Transfer

The RM requires a FileCopier to efficiently and securely copy files from one Grid location to another, similar to file transfer services such as the conventional FTP service. This is a low level service with a simple interface and no smart behavior.

We plan to use GridFTP as the underlying transport.

4.2.5.2. Policies

Certain policy issues are 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?

4.2.5.3. Replica Consistency

Maintaining consistency among replicas in different sites and storage locations is a task for a Consistency Service (to be defined) on top of the ReplicaManager. The Consistency Service will deal with updates of replicas and guarantee 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.

Preliminary models for replica consistency can be found in [R18]. More research is necessary here which will then lead to a detailed design.

4.2.5.4. Synchronization

The RM is also responsible for keeping the RC 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 is removed without the RM's knowledge. To handle this problem, the RM will periodically ask the SE if the replica is still there.

In contrast to a database management system (DBMS) where the DBMS has full control over all read and write access to data, the Replica Manager does not have control over files in the SE. Thus, several problems and inconsistencies can occur and detailed policies to deal with possible inconsistencies have to be worked out. In addition to periodically checking files for availability (which might be difficult since files may be at tape rather than disk locations), if a client request for a local file (registered in the RC) fails, the corresponding file entry can be deleted from the RC.

4.2.6. Replica Selection & Cost Estimation

A replica selection service selects the "best" physical file based on a given logical filename and a storage destination. Several performance parameters (like network speed, current network throughput, load on data servers, input queues on data server etc.) have to be included into the metric for selecting the "best" replica. Since currently no database management system is providing such a service, further investigation and research on our part is necessary. Only future versions of this document will therefore contain a detailed design for Replica Selection. Collaboration with the other work packages on this issue, as well as further input from the applications, is needed.

4.2.6.1. Cost Estimation

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.

4.3. SECURITY

4.3.1. Introduction

Systems requiring protection of information are encountered every day spanning a wide range of needs for organizational and personal privacy. However, a generic and deep discussion on computer security (understood as the set of techniques that control who may use or modify the computer or the information contained in it) is out of the scope of this document. Instead, we will describe the basic principles related to information protection and then propose some security enhancements for some components that have been defined within our architecture.

Historically, security specialists [R22] have found it useful to place potential security violations in three categories:

- Unauthorized information use: an unauthorized person is able to read and take advantage of information stored in the computer. This category of concern sometimes extends to "traffic analysis," in which the intruder observes only the patterns of information use and from those patterns can infer some information content. It also includes unauthorized use of a proprietary program.
- Unauthorized information modification: an unauthorized person is able to make changes in stored information - a form of sabotage. Note that this kind of violation does not require that the intruder see the information he has changed.
- Unauthorized denial of use: an intruder can prevent an authorized user from referring to or modifying information, even though the intruder may not be able to refer to or modify the information. Causing a system "crash," disrupting a scheduling algorithm, or firing a bullet into a computer are examples of denial of use. This is another form of sabotage.

Also known as “countermeasures”, there is a basic set of defenses such as authentication, authorization, cryptography, and intrusion detection. Authentication is the process to verify the identity of a person (or other agent external to the protection system) making a request and authorization is the process to grant a principal access to certain information.

4.3.2. Cryptography

We present here an overview of the aspects of interest for systems security. For a more detailed discussion see [R23, R24]. Basically, encryption is encoding a message to hide its meaning and is a valuable mechanism that can be used to provide:

- Authentication: Can authenticate the identity of users, transactions, and systems.
- Protection of messages: Can protect the secrecy of a message and prevent illegal modification. Cannot protect against destruction of the message.
- Protection of software and data: Can protect the confidentiality of them although not avoid their destruction. For example, passwords can be encrypted.
- Digital signatures: Can authenticate the origin of a message.
- Nonrepudiation: A user that signed or otherwise authenticated a document using cryptography cannot deny having sent it.

We can classify cipher systems according to three independent aspects [R23]:

- Number of keys used: symmetric (one key) and asymmetric (encryption and decryption keys, these are the public-key systems). Neither approach is the best for all cases.
- Type of encrypting operations: Symmetric systems use substitution and transposition stages. Substitutions just replace a bit or character for another. Transpositions rearrange bits or characters in the data. Product ciphers are combinations of substitutions and transpositions. Public key systems are based on invertible mathematical functions.
- The way the plaintext is encrypted: block and stream ciphers. In a block cipher a block of data is transformed, using a key, into a block of ciphertext. In a stream cipher a stream of key bits is used to encode a stream of data one bit or character at a time. Block ciphers are more appropriate for use within computers, while stream ciphers are seen mostly in communications. Block ciphers can simulate stream ciphers and we consider only this type here.

In the context of our work we will be interested in public key systems since these are widely used and because Grid software based on Globus uses this approach for data encryption. Public key systems use two keys, one of which is public and the other secret. The approach is based on the infeasibility of determining the decryption key given the algorithm and the public key. Instead of permutations and substitutions these algorithms use properties of mathematical functions. In particular, they use the theory of NP functions, those for which there is no polynomial time algorithm.

The public keys are normally registered with a Certification Authority (CA). This authority distributes certificates, which are public keys with the signature of the CA. There are authentication and attribute certificates. Attribute certificates assert that certain properties are true of the owner of some authentication certificate. Attribute certificates are used in SSL and other protocols. A certification authority is not a problem for an institution but it is difficult to find CA's that are acceptable to a large number of institutions across states and countries. Another problem is that encryption is slow. There is no way to improve the security of a PK system by increasing the size of the key because they depend on how hard it is to solve a mathematical problem.

4.3.3. Security Models

Because of the importance of creating a suitable security system for the end-users, it is important that designers of such systems follow the basis of generalized models with well known properties. We will briefly describe here some of the proposed models. Models can be discretionary or mandatory. In a discretionary model, holders of rights can be allowed to transfer them at their discretion. In a mandatory model only designated roles are allowed to grant rights and users cannot transfer them.

4.3.3.1. Access Matrix

The model defines a set of subjects (requesting entities), a set of security objects (requested entities), and a set of rights or access types (the way in which the object can be accessed). In the original matrix of Lampson [R27], there was the concept of owner, which as we have discussed in the last chapter, is a violation of the principle of separation of duty. There is also the concept of "controller", with special rights. The Lampson matrix include operations to modify the matrix and to allow propagation of rights. These are "administrative" operations because they would normally be used by a security administrator. An implementation of the access matrix must have a way to intercept user or program requests and compare the request to the access matrix to decide access.

4.3.3.2. The Bell-Lapadula Model

The Bell-LaPadula Model (BLM), also called the multi-level model, was proposed by Bell and LaPadula [R28] and it supports hierarchical access control. In this model based also on an access matrix subjects and objects are partitioned into different security levels and a subject can only access objects at certain levels determined by his security level. The Bell-LaPadula model supports mandatory access control by determining the access rights from the security levels associated with subjects and objects. It also supports discretionary access control by checking access rights from an access matrix.

4.3.3.3. Role-Based Access Control (RBAC)

RBAC can also be seen as a variation of the Access Matrix model [R29] where subjects can only be roles. RBAC is a form of mandatory access control, but not based on multilevel security requirements. Rather, access control decisions are determined by the roles individual users take on as part of an organization. In other words, rights and permissions are assigned to roles rather than to individual users and they acquire these rights and permissions by virtue of being assigned membership in appropriate roles. Major advantages to RBAC are flexibility and low overhead. Flexibility allows for the enforcement of least privilege, conflict between duties, dynamic and/or static separation of duties. It allows for minimal effort in allowing and revoking the user's role based on job and responsibilities and decentralization of administrative tasks is also made possible.

4.4. OPTIMISATION

We define ***Grid Query Optimisation (GQO)*** as the optimisation of the time and/or cost of execution of a Grid job submitted by a user while performing analysis. In the following the terms job and query will be used interchangeably.

We can consider query optimisation from various points of view:

- **User oriented optimisation (high performance computing)**. Each physicist would like to perform analysis on HEP events as fast as possible and possibly minimizing the cost for execution of jobs that she submits to the Data Grid. The optimal situation for a physicist would be being the only user of the Data Grid, thus having all the Data Grid resources personally available.

- **Grid oriented optimisation (high throughput computing).** Grid designers have a different perspective on the Data Grid. Their task is to define optimisation services that "make happy" all the users of the Data Grid, without favouring any of them (of course, different categories of users can be given different priority of use of the Data Grid). Thus, collective optimisation should be taken into consideration, trying to maximize exploitation of Grid resources while maintaining acceptable time/cost for execution of single physicist jobs.

- **Site oriented optimisation.** Administrators of Data Grid sites would like to decide upon local policies of use for resources at that site. These policies will influence the usage of the site by Grid and non-Grid jobs.

The optimisation services provided by the Data Grid should be designed to take into consideration all three of these perspectives and the "right" trade-off between them. We can consider two kinds of optimisation:

- **Short-term optimisation of user queries.** This optimisation can be activated whenever a query is submitted to the Data Grid and aims at minimizing the execution time and cost of the job. Optimisation is based on the information specified by the job: input data set and job code. We identify some possibilities for optimisation:
    - **Job decomposition.** A job usually operates on a set of events independently. This means that a job can be decomposed as a set of sub jobs, whose code corresponds to the code (or part of the code) of the main job, and an aggregation job, that executes on the output data set of the sub jobs. Job decomposition can be performed on the basis of where the input data products are stored in the Data Grid, or on the structure of the job code.
    - **Job execution time estimation.** This information could be exploited by the user in order to decide whether or not to submit a certain job to the Data Grid, according to how long she is willing to wait for job completion. Execution time execution can be based on both the location of data and/or code execution time estimation for each of the sub jobs that compose the job.
    - **(Sub)Job dispatching.** Where to dispatch a (sub)job is another important issue to take into consideration for GQO. The optimum location for the execution of a job depends on the location and current status of both the required data and the required computation resources. A trade-off between data optimisation and computation optimisation is important to assure Grid oriented optimisation. As far as data optimisation is concerned, there are two important aspects
        - *Data selection.* The same input data product for a job could be replicated in several files, placed in different locations on the Grid. The most convenient file copy should be selected for use by the job.
        - *Data relocation.* The relocation and replication of files inside and between sites must be performed in an optimal manner.
    - **During execution optimisation.** When a job requests a set of data that is not available on the site where the job is running, a decision has to be taken on how to optimally access the missing data. Therefore, data selection and relocation could also be performed during the execution of the job, to meet the optimisation needs that arise due to unforeseen data requirements.
- **Long-term optimisation of use of Data Grid resources** As previously stated, the optimised use of Grid resources improves the overall performance of the Data Grid and thus, on average, the performance of jobs submitted by single physicists. Long-term optimisation can be based on statistics on the past use of the Data Grid and forecasts of its future use.
    - *Replication.* A possibility of long-term optimisation is to set up a suitable replication policy for files stored in the various sites of the Data Grid. For example, suppose that several jobs submitted to the Data Grid from sites placed in the same area have been accessing much the same set of files. Then, an immediate optimisation is to replicate those files in a site easily accessible from the sites in that area. Analogously, if a set of jobs to be executed in the future in a certain area of the Data Grid is going to use a similar set of files, these files could be replicated in advance and stored in sites included in that area.

Reclustering. Input data products of a job might be sparsely spread over several large files. Reclustering them into a smaller set of files prior to their analysis could improve execution time of the job, and more fully exploit the content of the files. A possibility is to set up some *reclustering policy* for data products. For example, if several jobs are going to access almost the same set of data products in the future, it could be convenient to store these data into the same file (or a set of files).

5. DETAILED DESIGNS

5.1. THE DATA REPLICATION SYSTEM

The Data Replication System is composed of several Grid Services, such as the ReplicaCatalog and ReplicaManager. We will try to discuss the System in the context of the full DataGrid design, not just WP2. We try to identify each necessary service and put them into detailed context, like which node they can be run on, how they interoperate etc.

The **Data Replication System** needs several capabilities, including:

- Maintaining a consistent and scalable **Replica Catalog**. The ReplicaCatalog service 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 Replica Catalog. 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, file type and file size.

Our main focus is on **file** replication. File vs. object replication is discussed in greater detail in the section on optimisation.

5.1.1. Replica Catalog

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, file type, etc.). The ReplicaCatalog 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 ReplicaCatalog API (e.g. using a RDBMS, ODBMS or LDAP) can be used without introducing interoperability problems.

There are several ways to implement a replica catalog based on how one decides to distribute or partition replica catalog data. We can identify three sample catalog distribution options:

- **Central replica catalog service:** A centralized service and thus a central data store contains all catalog information and is contacted for each LFN to PFN lookup. *Pros*: No synchronization problem, because all files are kept in one catalog. Easy to administer. *Cons*:

Not scalable, potential bottleneck due to WAN latency and high traffic. Single point of failure. Remote organizations are dependent on central organization for administration.

- **Replicated catalog service:** Each site has a replica catalog that keeps a list of all locally and remotely available files. All catalogs store identical data and are kept consistent under update. *Pros*: Scalable (assuming low update rate) because load is spread among multiple catalog instances. Access is localized, hence latency minimized. No single point of failure. *Cons*: Hard to administer, to implement efficiently, and to synchronize. Potentially stale data due to lazy synchronization semantics.

- **Partitioned catalog service:** Each site has a replica catalog that contains a list of local files only. Catalogs are independent and autonomous, but linked together through a hierarchy or graph. *Pros*: Scalable because load is spread among multiple catalog instances. Access is localized, hence latency minimized. There is no single point of failure. Local and independent administration. *Cons*: Must traverse hierarchy to locate data. Potentially complex update propagation. Potentially stale data due to lazy synchronization semantics.

We believe a variant of the third option (partitioned catalog service) maps best to the requirements of a replica catalog. The replica catalog design we propose is a hierarchical system with interconnected parent and child catalogs, forming a distributed tree or graph (see Figure 5.1) which can be traversed. We assume that replica catalog sites do not need to be synchronized immediately, but only within a specified timeframe of, for example, several minutes (i.e.: relaxed consistency model). We distinguish non-leaf Replica Catalogs and leaf Replica Catalogs as follows:

- 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 attributes include file size, time-step, check sum, creator, master/replica, owner, file type 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 catalogs do not have physical file information but only store the URL of child replica catalogs. A non-leaf catalog redirects requests for a physical filename to children, which may in turn propagate the request until all physical file information is found. The additional redirection lookup steps are transparent to the end user. Note that preliminary work on replica catalog redirection has been presented in [R17]. 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 the following. A leaf replica catalog is co-located with each StorageElement. We refer to these catalogs as StorageElement-ReplicaCatalogs (SE-RC). If a site has multiple SEs they may choose to have a non-leaf replica catalog which acts as a site RC whereas small sites may choose to omit a site RC, and only use SE-RCs (see Figure 5.1).

To summarize:

- 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 Virtual Organization.

- 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 5.1)).

The top level RC will be heavily used and thus needs to be replicated for two classical reasons:

- availability - one server could be down and thus the top level information is not accessible
- performance gain for response to read requests (load balancing)

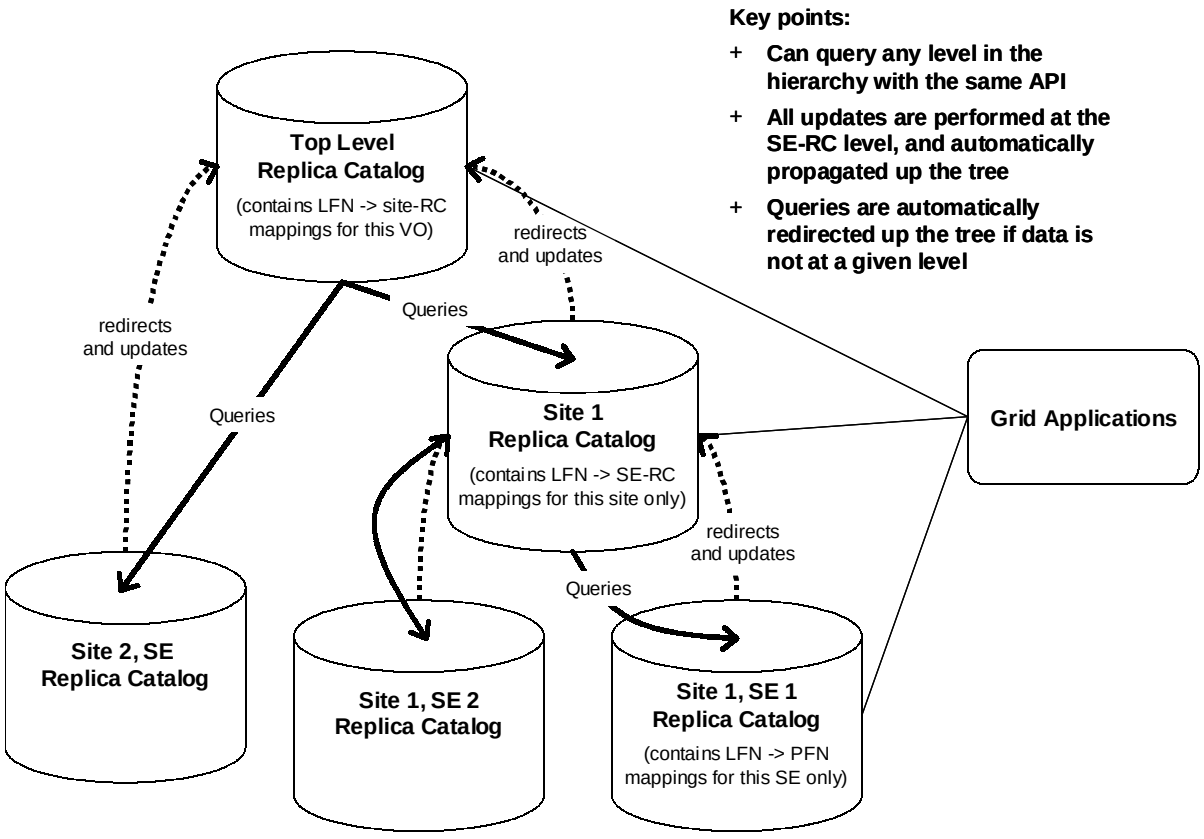

Figure 5.1: Distributed Hierarchical ReplicaCatalog

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 (i.e. to the site RC, which in turn will periodically send 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 users know 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 affecting 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.

The RC API is described in Chapter 6. Note that the ReplicaCatalog service is mainly used by the ReplicaManager.

5.1.1.1. Replica Catalog Deployment Example

As a concrete example, consider one potential structure for the CMS experiment (see Figure 5.2): 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). In addition, there will be a single, global Root Replica Catalog at CERN.

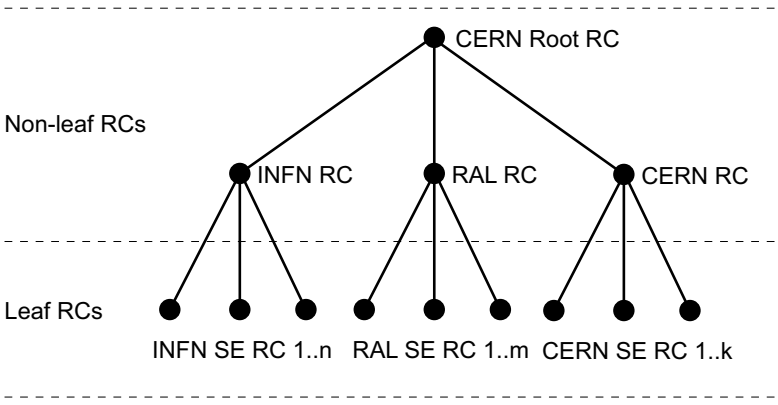

Figure 5.2: Outline of hierarchical, distributed replica catalog

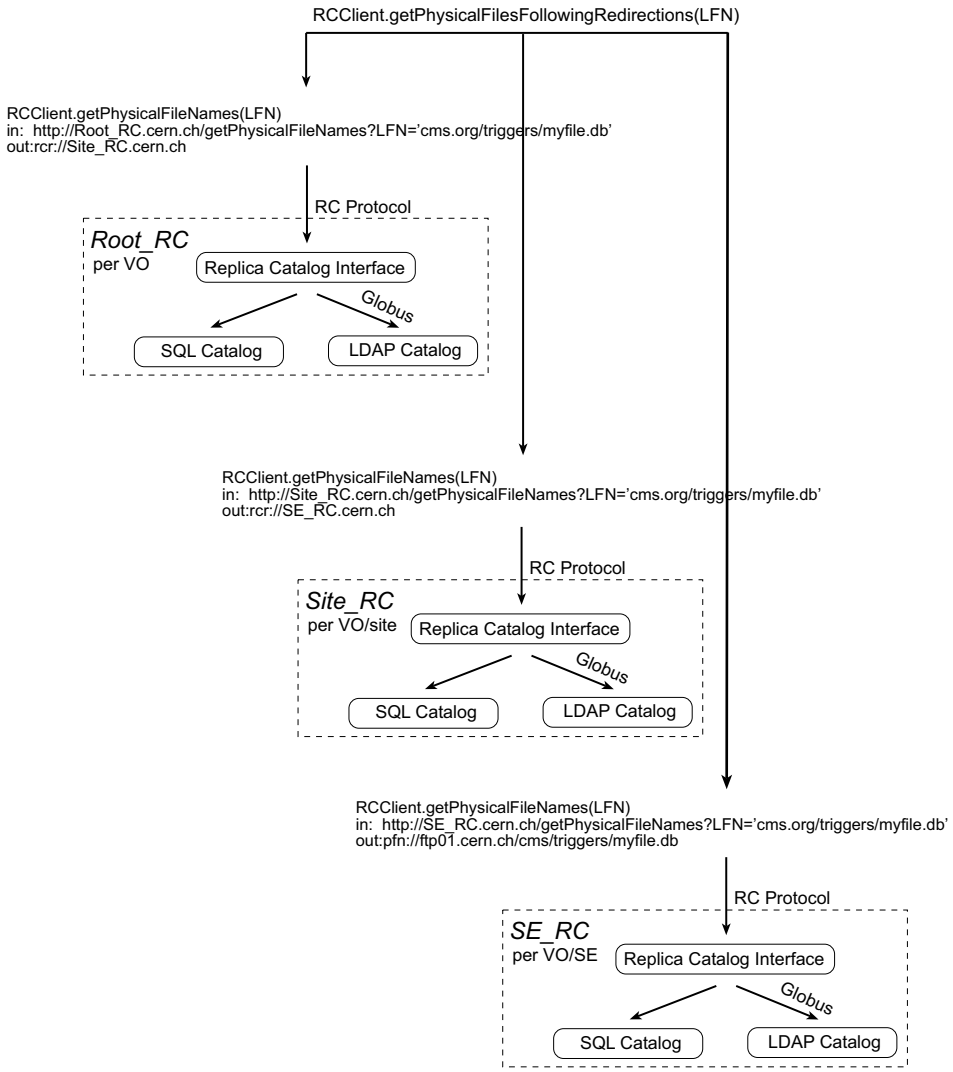

Figure 5.3: Interaction in hierarchical, distributed replica catalog

We assume that the protocol to talk to the Replica Catalog is HTTP. In the following example (see Fig. 5.3) 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, 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 a 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 is 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 5.3) 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 catalogs. This information is required for loosely coupled batched update notification whenever entries are added/deleted to/from a leaf replica catalog. 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 catalog approach can be summarized as follows:

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

5.1.1.2. Replica Catalog Sample Usage

A new replica entry is added directly to the leaf catalog (SE-RC), thus ensuring that the local catalog is always up-to-date. Periodically (e.g. every few minutes), the SE-RC propagates updates up the tree, for example to the site RC, which in turn periodically propagates updates to the top level RC. When doing a lookup of all PFNs for a given LFN, a Grid Scheduler (or a user application) can query any RC. If desired, the replica catalog can automatically redirect requests up and/or down the tree until all possible PFNs are found. If the user knows that a replica is (or was) at a given site, they can improve performance by directly querying the RC of that site or SE. If the replica is no longer there, the request is redirected back up the tree. This way the load is spread over many catalogs, relieving the top-level catalog of having to satisfy all requests. We assume that applications are mostly interested in information on local files, so tree traversals are rare. A runaway query can be prevented by specifying timeouts. In addition, a query can be scoped to only contact a given set of replica catalogs. This also provides fault tolerance. The site RC or the top-level RC can be down or unreachable without affecting local operations. The top-level server, which contains an entry for each LFN in a given virtual organization, should be replicated for increased fault tolerance and scalability.

5.1.2. Replica Manager (Prototype: GDMP)

GDMP [R8,R11] originated as a pilot project to test data management issues on the Grid. Its purpose was initially to replicate Objectivity database files across Grid nodes. GDMP is already implemented and works, unlike other services we plan to provide. The services that follow will benefit from the experience of GDMP and will eventually replace it. GDMP is one of the major components of the Month 9 deliverables of WP2. GDMP implements most services itself in some ways internally, which we will extract and enhance into the 'real' services. The next version of GDMP (i.e. after the Month 9 deliverable) will then be using those services as an application to achieve the same functionality and more.

GDMP is a file replication software system initially designed to replicate Objectivity database files from one site (storage location) to one or more other remote sites. In the current version, arbitrary file types (e.g. ROOT files, ZEBRA files, etc.) can be replicated from one site to another. GDMP works as follows:

A site produces a set of files locally and another site wants to obtain replicas of these files. In the case of Objectivity files, each site is running the Objectivity database management system locally, with its own internal catalog of database files. However, the local Objectivity database management system does not know about other sites and a replication mechanism is required that can notify other sites about new files, efficiently transfer the files to the remote site, and integrate the filenames into the Objectivity internal file catalog. An additional server needs to be available at each site to handle replication requests and to trigger file transfers, notification messages, and updates of local catalog information. Simply put, this is done by a GDMP server running at each site where files are produced and possibly replicated. In case of ROOT files (or any arbitrary file type), a local site catalog is managed before files are replicated to other sites.

The GDMP replication process is based on the producer-consumer model: each data production site publishes a set of newly created files to a set of one or more consumer sites, and GDMP ensures that the necessary data transfer operations (including all the steps mentioned above) complete successfully. These services are implemented by a set of interacting servers, one per site participating in the data replication process. In summary, GDMP client command line tools provide four main services to the end-user:

- subscribing to a remote site for getting informed when new files are created and made public,
- publishing new files and thus making them available and accessible to the Grid,
- obtaining a remote site's file catalog for failure recovery, and
- transferring files from a remote location to the local site.

GDMP uses the Replica Catalog to for publishing replica information. For more details on GDMP and the usage please refer to the **GDMP User Guide** [R10].

5.2. SQLDATABASESERVICE

The SQLDatabaseService allows convenient, 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, and a programming language API. A well defined language, platform and RDBMS neutral network protocol between client and server is used. Thus, the service can be seen as a unified Grid enabled front end to relational databases. It can, for example, be used from C/C++, Java and Python to access data held in local or remote MySQL, Oracle, DB/2 or Postgres databases, with clients using Linux or Windows and servers running Solaris. 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 Java 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). It then dispatches the SQL query to a SQL database backend tier for data storage and retrieval. Finally the server side logic trivially converts the SQL result 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 this nature.

5.2.1. Component Design

The main component in the proposed design is a servlet that processes SQL queries and outputs the result set as XML. Queries to retrieve, insert, or delete data are embedded within XML templates pages (XSQL files). These pages are accessed over HTTP. The user enters a URL through a browser or programming API, which is interpreted and passed to the servlet through a Java Web server servlet engine. The URL contains the name of the target XSQL file (.xsql) and, optionally, parameters such as values and an XSL stylesheet name. After some processing, the servlet is able to retrieve the SQL queries from the page, connects and sends the queries to the underlying database which returns the query results. Figure 5.4 depicts an interaction diagram of a complete request-response call chain (where responses are implied by the usual convention).

Figure 5.4: Interactions of SQLDataBaseService

The service uses the following mechanism in each respective domain:

- **Network transport protocol for client server communication**: HTTPS (i.e. http over SSL/TLS). Reason: A reliable request-response protocol is sufficient. https allows to leverage the massive existing infrastructure of flexible, robust and scalable, easy to use http based tools. Vendor and programming language restrictions are 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 and 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 optimised 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, X.509 Certificates (GSI compatible). Reason: Simply the only viable options.

5.2.2. Examples of Use

We now demonstrate sample usage of the service through two example use cases. The first use case describes how to retrieve metadata about logical file names from a replica catalog. The second use case shows how to insert logical file names and associated meta data into a replica catalog. Let us assume we have a relational table “Rccatalog” with LogicalFileName and filesize columns:

```
Table RCcatalog
LFN                           | Size
--------------------------------------
host1.cern.ch/somepath/file1 | 10000000
host2.cern.ch/somepath/file2 | 50000000
```

The SQLDBservice supports any number of user defined queries. A query is defined via a template file. The template file for a "getLogicalFileMetaData" query contains a "`select * from table`" SQL retrieval statement. More precisely, it reads as follows:

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

For our discussion, the first and last line contain syntactic sugar and can safely be ignored. As can be seen the query contains a variable "@catalog". Variables are later substituted with actual values upon query invocation. In order to retrieve the metadata associated with all known logical file names, a client invokes the query by sending query template name, variable names and variable values to the server. This is done by issuing a HTTP GET request, either from a Web browser or client API, to the following URL:

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

Upon invocation, the query variable "@catalog" is substituted with its actual value "Rccatalog". Next, the resultant query "`select * from RCcatalog`" is executed against the database 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>
```

Canonical XML defines a direct mapping from SQL to XML and vice versa: 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. For further details consult [R16].

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 request 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>
```

The "myinsert" query template file is defined with the appropriate insert command:

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

5.2.3. Deployment

Due to its flexible nature, the SQLDatabaseService can be customised heavily without a need to compile or write any code. The service establishes persistent connections to backend RDBMSs based on a configuration file containing information such as JDBC database driver names to be used, database hostnames and ports, database users, passwords, etc. Query template files are not hardcoded into the server. Instead, they are dynamically picked up from the file system (just like HTML pages in normal Webservers). Thus one can easily add, remove or modify queries via intuitive file operations.

The service can be run in a small environment with moderate performance and availability requirements. It can, however, also be deployed to transparently and reliably handle hundreds to thousands of concurrent requests per second. A scalable high availability deployment scenario of a single SQLDatabaseService is shown in Figure 5.5 The setup has N concurrent clients, M http servers (e.g. Apache), O servlet engines (e.g. Tomcat), and P RDBMS instances (e.g. Oracle Parallel Server). Each of these can (but need not) run on a different box. Improved performance is achieved by

transparent load balancing over all these levels, as well as thread, connection and transaction pooling. Transparent failover is implemented via IP redirection. Note that this setup requires no complex request routing on the client side, because it simply appears as a single large service to clients. Also note that for simplicity and performance reasons, all constituents of the service should reside on a low latency high bandwidth LAN.

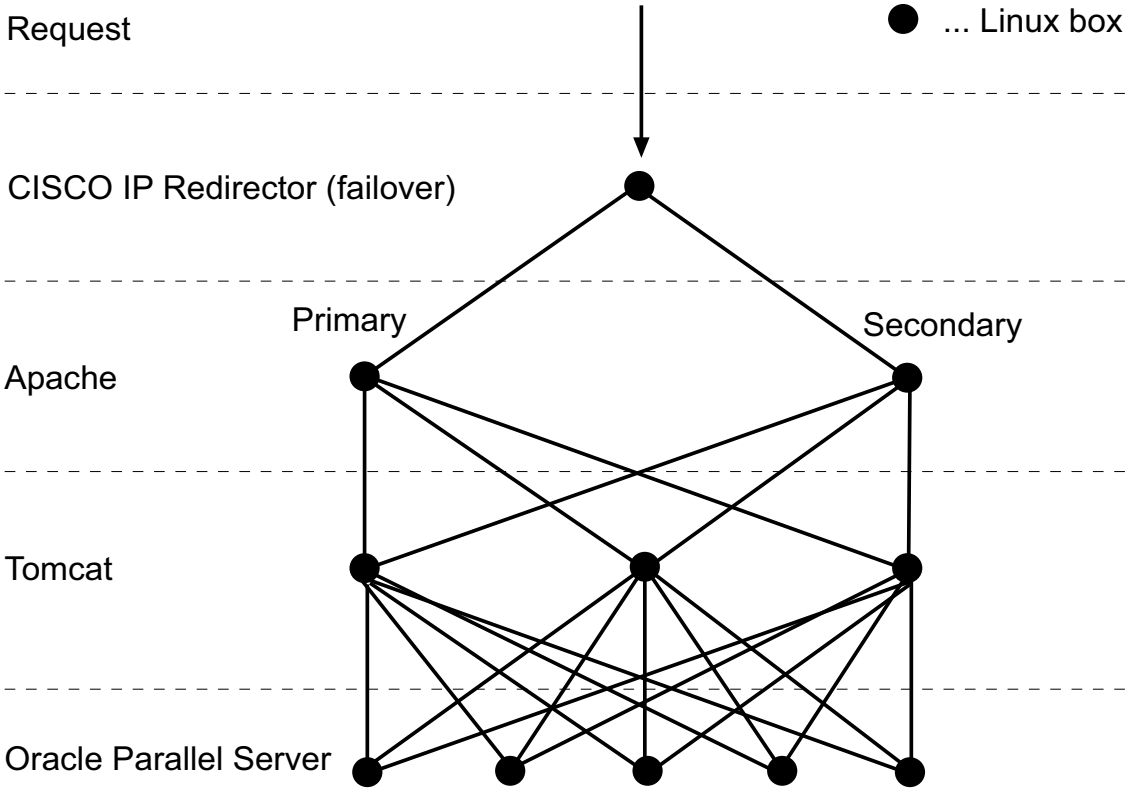

Figure 5.5: Scalable high availability deployment of SQLDatabaseService

5.3. SERVICEINDEX

Service indexes provide the glue that ties together a large number of decentralized Grid components to support collaborative work within virtual organisations. For example, Grid services are utilized for service discovery, job submission, network, systems and application monitoring as well as software and system configuration management. At the core, Service Indexes connect loosely coupled information providers and consumers and enable desired communication flows in a secure, efficient and flexible manner.

A Grid consists of one or more *information spaces*. An information space is an abstract metaphor and contains the structured or semi-structured meta data needed to carry out a given application use case. For example, job submission requires information about the program being submitted (e.g. estimated CPU/IO ratio, number of threads and processes, interactive or batch) and the resources necessary to execute the job (e.g. architecture, memory, disk and tape storage, dependencies on shared libraries and versions, availability and connectivity to local and remote third party services like database engines, ftp, http, tape stagers, AFS/NFS). An information space is constructed from a number of information sources (service providers, henceforth called *services*) which are network connected with information sinks (consumers, henceforth called *clients*). More precisely, a service is a continually running network attached program waiting to serve client requests (daemon). Clients and services are

communication end points (peers) and speak platform and language neutral network protocols, hence can reach across architectures and organisational boundaries (subject to local security policy). Clients may themselves be services, in which case they play the role of proxies or mediators, making available a custom tailored structured view of the underlying information space. For example, search engines and service brokers delegate requests to one or more other services and integrate their results. Proxies can be stateless or maintain memory or disk resident soft state (caches). Note that there neither exists a single grand monolithic database containing an information space nor a single owner thereof.

This model is similar to a network of http servers serving static or dynamic HTML pages linked by URLs and being consumed by human clients. The Web model is a powerful basis to leverage from.

5.3.1. Scalability Requirements

Grid projects are often undertaken as collaborations of multiple large and small organisations, and typically based on distributed n-tier models. Such systems cannot be managed centrally. For example, database catalog services, replica management services, file transfer services, software and system configuration management services, as well as monitoring services may run independently on N>10 sites and M>10000 nodes. Consequently, many services are partitioned and/or cloned, both among institutions and within a site, which in turn exposes many different kinds of scalability limits, and can lead to communication boot-strapping and systems management problems. We are not concerned with small scale ad-hoc solutions that work fine on the department level but break down on the global scale, for reasons including inflexibility, implicit knowledge, efficiency, reliability and manageability. Instead we attempt to tackle the fundamental mechanisms that turn ad-hoc information services into Grid Information Services.

Before a client can ask a service to execute a request on its behalf, it needs to get a "handle" to the service in the first place. More generally the following basic questions arise:

- How do I find a service to which I can ask the questions below? This is termed the *service boot-strapping problem*.
- Which services are available at any given moment in time within a virtual organisation or a subgroup thereof?
- On which hosts and ports do these services live, what protocols do they speak and what *kind* of information do they offer to whom?

In a small, centralized environment such questions can easily be answered. Implicit knowledge ("everybody knows that `afs43.cern.ch` is a semi-public Sun450 fileserver with 0.5 TB hot-swapable SCSI raid disk"), or a phone call, or consultation of a centrally managed configuration file yields answers. Decentralized Grid environments call for more scalable solutions.

5.3.2. Design Principles

A service index service (SIS) infrastructure is purely concerned with enabling (rather than replacing) communication between peers. It assists a client in finding a service matching its needs. The client can subsequently direct more elaborate and specific questions to the matching service. At the core, a distributed SIS infrastructure keeps track of a graph of services and associated descriptions, which can be traversed, i.e. searched. Figure 5.6 depicts a graph of services.

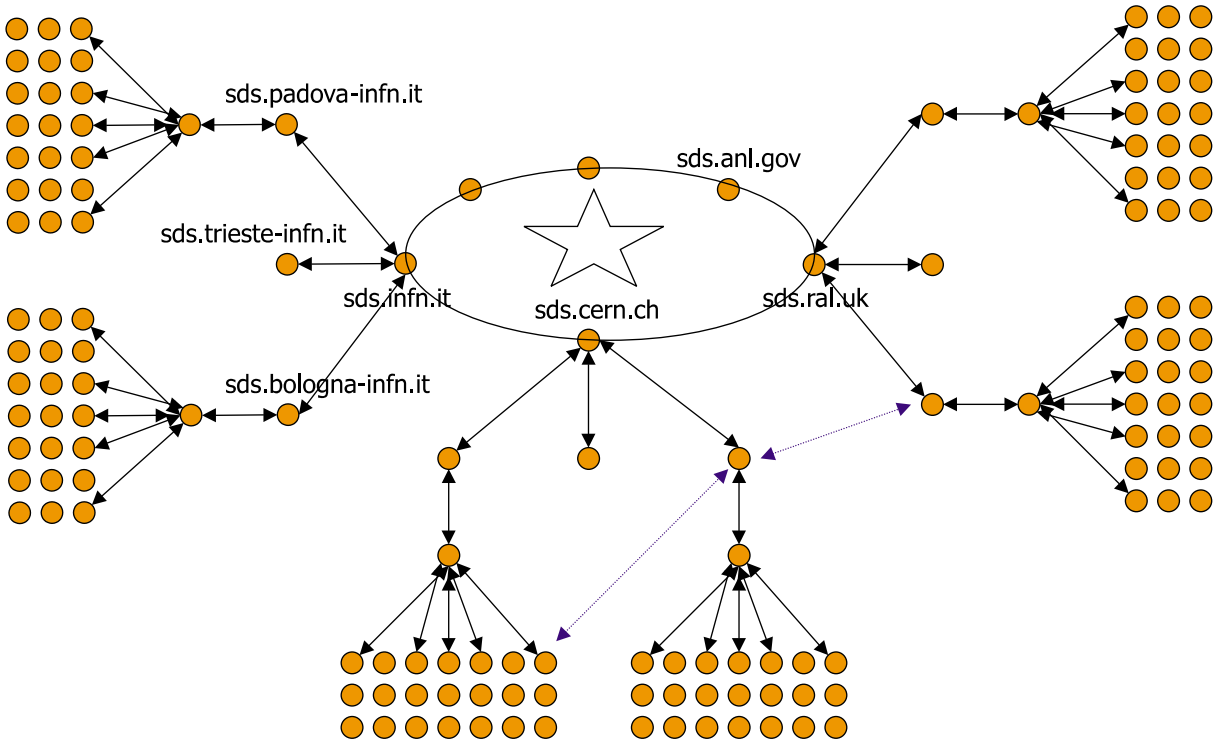

Figure 5.6: Global Graph of Services

A graph is favoured over a rigid hierarchical structure, because a single hierarchy cannot reflect equally valid decompositions such as by network topology, ownership, or functionality. This is somewhat similar to a Web infrastructure maintaining a graph of hyperlinked HTML pages, which can be navigated. The SIS keeps track of changes over time, as services crash, are added, reconfigured, or phased out. Note that in the core SIS model (phase 1), there exists no "intelligent" search engine and query language. The only means to discover matching services is by traversing (crawling) the graph. The core model is limited in usability with respect to the true desires of clients, yet provides the foundation on top of which more advanced search engines or custom tailored matchmaking brokers can be built (phase 2). Note that undertaking phase 2 before or without phase 1 means constructing small centralised information islands with little chance for integration into a "Grid".

The graph ensures that for each client end point there exists at least one path to *reach* every service description, possibly via several hops. Assuming the graph is in place, then a client is confronted with the problem of finding the appropriate entry point into it. An organisation may for example choose to provide its clients a top level entry point into the global service description graph. This top level entry point (Root SIS, or simply RSIS) can either be a well known host through conventions (e.g. `rsis.localdomain:55555`) or dynamically discoverable through DNS SRV records. In the latter case, a query to the local domain's DNS server will return the address of the RSIS (this is the way the Internet email infrastructure scales and works). Let us now examine in more detail the properties of services, descriptions and SIS's.

Service. To allow construction of graphs, a service keeps a list of *parent SISs* to which it periodically (re)announces its description in order to provide for simple and fail-safe change propagation. A service supports a protocol that returns its service description as well as its parent service descriptions upon request (subject to security policy).

Service Description. With each service a "Service Description" is associated, containing the information a client needs in order to be able to effectively communicate with the service. The description contains mandatory and optional items. Mandatory items require collaboration-wide standardisation to achieve interoperability. Descriptions are defined in XML. All descriptions inherit from a standardized base XML schema, possibly using the Web Service Description Language (WSDL). Therefore, the base schema can be extended in a type-safe manner.

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 of interest. 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>
```

In order to avoid overloading the service index infrastructure, i.e. to keep it efficient, reliable and operational, it is highly recommended to include optional information only if it is both light-weight and necessary for fast filtering in high level searches. More detailed information would better be obtained by querying the replica service through its private "replica-manager" protocol or through custom tailored search engines, both integrating service descriptions and results of queries to the replica managers itself.

Service Index Service. An SIS *is a* service, and as such inherits all characteristics of a service, including the fact that it can have parents and periodically (re)announces its description to them. To allow construction of graphs, it additionally keeps a list of children services (or more precisely, their service descriptions). An SIS supports a protocol that returns its children's service description upon

request (subject to security policy). Children can register themselves through an announce-listen protocol (subject to security policy). In order to allow for fail-safe, simple and automated distributed garbage collection, service descriptions of children are kept as *soft state*, that is, they are cached for a limited amount of time. In other words, service descriptions are tagged with time-to-live tokens (TTLs), hence expire and get dropped unless renewed via (re)announcement. Consequently, services can crash, stop, be added or changed without leaving stale state behind indefinitely. Soft state is desirable, because it elegantly avoids the countless complexities more elaborate designs face in attempting to determine when and why to remove stale data without compromising consistency and offending information producers. Hard state based distributed information systems populated from many independent sources quickly evolve into garbage dumps where valid information is hard to distinguish from trash, decreasing overall utility dramatically. If scalability and bandwidth are of concern, messaging protocols throttling announcement frequency can be used. Spammers ignoring throttling messages can be dropped without further notice.

5.4. SECURITY SOLUTIONS

5.4.1. A Security Model for the SQLDatabase Service

In our approach of providing secure access to services defined within our work package we will discuss now how this can be achieved for one of our key components, the SQL Database Service. This service has already been described but only a minimal part of its security requirements have been outlined. The proposed architecture for this service already copes with some basic security requirements but we will demonstrate that for a production level service, involving a large number of different Grid users, some additional requirements need to be supported. To understand what fundamental security requirements are already provided with the proposed design, we have to recall what are the main components and functionality provided by this service. Here we will just pay attention to those issues that are relevant for our discussion about security (a more detailed description of this service is given in a previous section in this document).

One could summarize the whole process by saying that the main component in the proposed architecture is a servlet that processes SQL queries and outputs the result set as XML. Queries to retrieve, insert, or delete data are embedded within XML datapages (XSQL files). These pages are accessed over HTTP. The user enters a URL through a browser, which is interpreted and passed to the servlet through a Java Web server. The URL contains the name of the target XSQL file (.xsql) and, optionally, parameters such as values and an XSL stylesheet name. After some processing, the servlet is able to retrieve the SQL queries from the page, connects and sends the queries to the underlying database which returns the query results. In order to achieve this, the XSQL Servlet uses a configuration file to access and authenticate the database connection.

A sample configuration file looks as follows:

```
<?xml version="1.0" ?>
<XSQLConfig>
        <connectiondefs dumpallowed="no">
                <connection name="demo">
                        <username>scott</username>
                        <password>tiger</password>
                        <dburl>jdbc:oracle:thin:@localhost:1521:ORCL</dburl>
                        <driver>oracle.jdbc.driver.OracleDriver </driver>
                </connection>
        </connectiondefs>
</XSQLConfig>
```

The XSQL Servlet finally expects to find an attribute named "connection" on the XML document's root element whose value must match the name of a connection defined in the configuration file.

5.4.1.1. HTTP over SSL

Since HTTP is the protocol used in our design to reach XSQL pages using standard URLs, it is easy to see that a straightforward access control mechanism can be implemented. One could use a secure extension of HTTP to restrict access to XSQL pages to authorized users only. Such a secure extension is already available and is widely known as HTTP over Secure Sockets Layer (SSL). The Secure Sockets Layer protocol is a protocol layer which may be placed between a reliable connection-oriented network layer protocol (e.g. TCP/IP) and the application protocol layer (in our case HTTP). SSL provides for secure communication between client and server by allowing mutual authentication, the use of digital signatures for integrity, and encryption for privacy (see [R20] for a detailed description). The protocol is designed to support a range of choices for specific algorithms used for cryptography, digests, and signatures. Choices are negotiated between client and server at the start of establishing a protocol session. This session is established by following a handshake sequence between client and server which may vary, depending on whether the server is configured to provide a server certificate or request a client certificate. HTTPS can be used in our service to authenticate clients (with X509 certificates) and deny access to those who are unable to present valid certificates.

5.4.1.2. Security Realms

Using HTTPS forces WEB servers to be modified to include SSL functionality and certificate management infrastructure in place to issue valid certificates to potential clients. Besides, it uses the URL scheme https rather than http and a different server port (by default 443). In those cases where such modifications are not possible, the use of security realms can be an alternative. A security realm is a mechanism used for protecting Web application resources. It gives the ability to protect a resource with a defined security constraint and then define the user roles that can access the protected resource. An example of a WEB server that has this type of realm functionality built in is Tomcat [R21]. It provides a mechanism by which a collection of usernames, passwords, and their associated roles can be integrated and used for access control. Tomcat provides two realm implementations called respectively memory and JDBC realms. Tomcat's MemoryRealm class uses an XML file as a repository for roles and user definitions. A sample memory realm XML file could be:

```
<tomcat-users>
        <user name="tomcat" password="tomcat" roles="tomcat" />
        <user name="role1" password="tomcat" roles="role1" />
        <user name="both" password="tomcat" roles="tomcat,role1" />
</tomcat-users>
```

Additionally, for every protected WEB resource, a collection of access constraints needs to be defined. This is achieved in Tomcat by adding security constraints (expressed in XML) to the Web.xml file associated to the resource. For instance:

```
<security-constraint>
<Web-resource-collection>
        <Web-resource-name>OnJava Application</Web-resource-name>
        <url-pattern>/*</url-pattern>
</Web-resource-collection>
<auth-constraint>
        <role-name>onjavauser</role-name>
</auth-constraint>
```

```
<login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>OnJava Application</realm-name>
</login-config>
</security-constraint>
```

This constraint specifies that everything under resource "Onjava Application" is only accessible by users playing the role "onjavauser".

JDBC realms on the contrary do not use an XML file to store user and roles definitions but rather a relational model stored in a relational database accessed with JDBC. This has the advantage that several servers can access a shared user-role database and modifications to this database are immediately visible by all the stakeholders. The only remaining issue to deal with is to tell Tomcat how to reach the user-role JDBC-enabled repository. This is done in a simple way by adding some configuration information. See an example below that uses MySQL as the underlying repository:

```
<realm
classname="org.apache.catalina.realm.JDBCRealm"
debug="99"
drivername="org.gjt.mm.mysql.Driver"
connectionurl="jdbc:mysql://localhost/tomcatusers?user=test;password=test"
usertable="users"
usernamecol="user_name"
usercredcol="user_pass"
userroletable="user_roles"
rolenamecol="role_name"/>
```

5.4.1.3. XSQL Basic Access Control

No matter whether HTTPS or security realms are used to restrict access to Web resources (in our case XSQL pages) it still remains to know what restrictions need to be enforced when accessing the underlying database system where SQL queries have to be executed. XSQL specification knows about what connections have to be established by binding a given XSQL page definition to a given database connection. This means that once a client has been granted access to an XSQL page she would be able to execute the SQL queries contained in the page playing the role of the user specified in the connection attribute of the root element in the page.

5.4.1.4. An Advanced XSQL Access Control Mechanism

The access control mechanisms described above are definitely valid for implementing basic level security, however there are some unresolved issues.

First, when using SSL, access control is enforced by inspecting the certificate that the client side presents during the handshake process. In this case access control policies are defined in terms of existing fields in the certificate (the validity period, the signing certificate authority, the distinguished name, etc). These fields can be even extended with X509 v3 extensions and most implementations of SSL provide mechanisms to access such extensions and implement complex policies. Though this is feasible, this access control mechanism is not bound to XSQL. This means that complex policies defined at the X509 certificate level cannot be propagated into database connections that are dependent on such policies. This also applies if security realms are used since information about roles and users defined in a JDBC realm is also not propagated.

Second, XSQL has an important restriction because each page is statically bound to a given connection type. Thus, if different clients would need to execute the same type of queries on different

databases we would need to replicate the required XSQL pages as many times as different types of connections would need to be supported (and this does not scale in a Grid environment).

To overcome these disadvantages we need a mechanism that, first, can generate dynamically (not statically bound to XSQL pages) different types of database connections, second that complex policies can be defined and third that we can leverage on existing technologies like X509 certificates. The work to be done to obtain an advanced XSQL access control mechanism will investigate the following components:

- A repository for X509 certificates with extensions that contains not only user information but also role and group relationships. X509 certificates are specially interesting for us as opposed to JDBC realms because the existing Grid middleware like GSI [R26] already makes use of this mechanism for security enforcement.

- An extension to the current XSQL tag-language to incorporate authorization information that can be dynamically translated into connection attributes within an XSQL page. This language extension should also leave the door open to mechanisms not based on X509 certificates such as Tomcat Realms.

- An extension to the XSQL servlet to process the language extensions defined above.

In general terms, and assuming X509 certificates as the mechanism to identify clients, the advanced XSQL access control mechanism would work as follows:

- The user enters a URL through a browser, which is interpreted and passed to the XSQL Servlet through a Java Web server. This is based on HTTPS and a client X509 certificate is requested (as part of the standard SSL handshake process)

- The XSQL page processor processes the requested XSQL page by looking for "Action Elements" from the XSQL namespace. The first action element to be processed being an authorization action element.

- The page processor invokes the authorization action element handler class to process the element.

- The handler obtains all the client information from the SSL layer and uses this information to obtain group and role based information from the authorization repository.

- The handler also obtains information about security constraints (allowed database connection types) for the accessed XSQL page based on the group, role, and user information previously obtained.

- The handler decides whether SQL query execution is allowed or denied for the accessed XSQL page and dynamically embeds within the XSQL page the appropriate database connection information.

The XSQL page processor continues the parsing of the remaining XSQL tags and the embedded SQL statements are executed with the database connection obtained from the previous steps.

5.4.2. A Security Model for CASTOR

5.4.2.1. Introduction

CASTOR [R4], the "CERN Advanced STORage manager", is a tape storage system developed at CERN. Its goal is to manage part of the huge amounts of information generated by the Large Hadron Collider (LHC) experiments starting in 2005. The aim is to manage this data in a fully distributed environment.

Today CASTOR is already in use at CERN for managing the information generated in some of CERN's current experiments e.g. ALICE, ATLAS, and COMPASS.

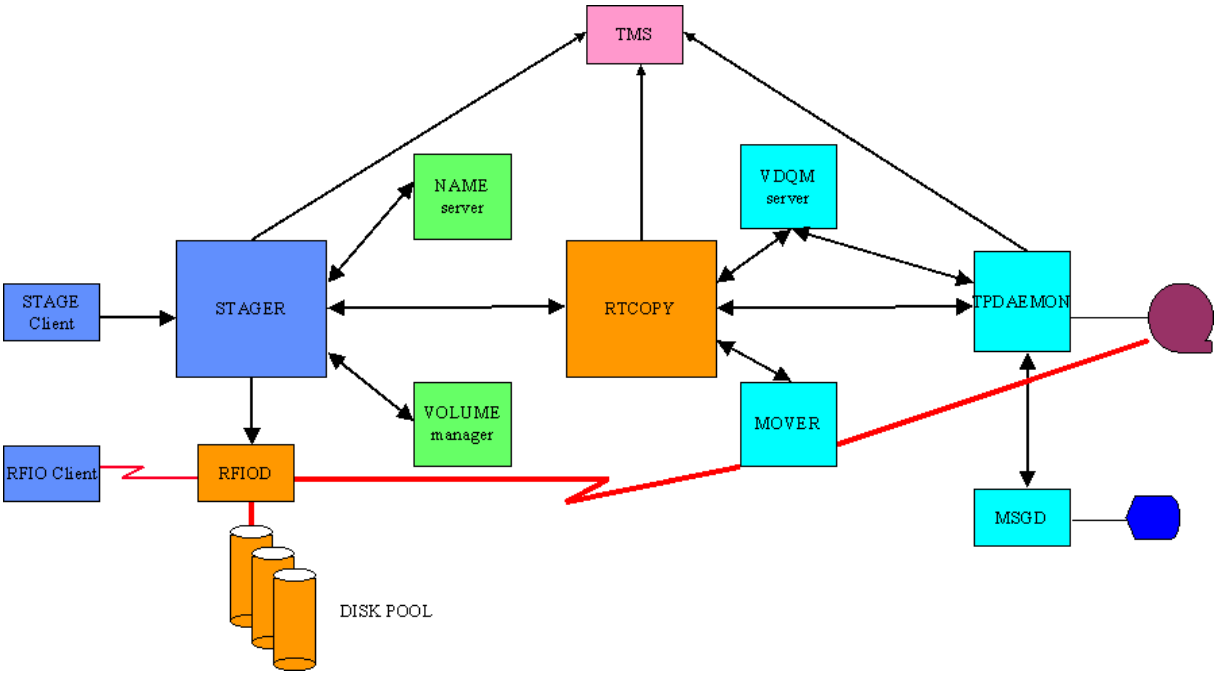

Figure 5.7: The CASTOR client server architecture

Today's version of CASTOR makes use of the host operating system's own security mechanisms for user authentication and authorisation. When an end user client program contacts CASTOR, the client program checks the end user's identifier and group identifier from the host it is running on. Next, it sends them as clear text to the CASTOR servers receiving the request. The CASTOR servers make authentication and authorisation decisions based on these two identifiers and then processes the request. See Figure 5.7 for details.

CASTOR does not support traffic encryption so all communications between the client programs and the server programs is in clear text.

There are several problems in the security model presented above. For example, an administrator of a host can impersonate an ordinary user and make the client program authenticate itself to CASTOR as if she were the ordinary user. In this way, an administrator may get unauthorised access to an end user's files stored in CASTOR.

The other problem is CASTOR's clear text protocols. An eavesdropper can find out what valid user identifiers and group identifiers exist by listening to the network communications. After intercepting an interesting pair, the eavesdropper can then construct the protocol elements that are needed e.g. to get a copy of a user's file stored in CASTOR, delete or modify one, etc. Man-in-the-middle attacks (where the attacker intercepts and possibly modifies information between two communicating peers) are also possible, resulting in the same kind of problems.

5.4.2.2. GSI and CASTOR

Due to the simplicity of exploiting the authentication and authorisation mechanisms of CASTOR, it was decided that the impersonation problem and man-in-the-middle attacks described earlier needed to be fixed.

Several options were studied. The first was Kerberos, which offers strong cryptographic services but suffers from a scalability problem effectively limiting its use to smaller organisations. In the case of CASTOR and CERN this is not feasible since scientific efforts at CERN usually involve several collaborating organisations and, in many cases, thousands of collaborators.

Several Grid middleware solutions were studied. It seemed that the Globus toolkit is emerging as the *de facto* middleware of choice for CERN. Among its components we find the Globus Security Infrastructure (GSI) module. This module, implementing the Generic Security Services Application Programming Interface (GSSAPI) standard, was found to have several interesting properties such as dynamic security algorithm negotiation, mutual authentication using X.509 certificates, support for several operating systems, etc. These characteristics fit in very well with the heterogeneous operating environment of the Grid and therefore that of CASTOR. Using X.509 certificates for mutual authentication will also prevent man-in-the-middle attacks.

After choosing GSI for enhancing the security of CASTOR, it was decided to split the work into two phases. The first phase involves using GSI to mutually authenticate end users to CASTOR’s front end servers. The second phase aims to improve the security of CASTOR’s internal servers by adding GSI mutual authentication between its front end servers and back end servers.

5.5. GRID QUERY OPTIMISATION

In this section we base our discussion on a Grid Architecture diagram (Figure 5.8) specialized to the "Grid Query Optimisation" (GQO) task. This is an extension to the description of the architecture in Chapter 3, i.e. Figure 5.8 is a specialization of Figure 3.1.

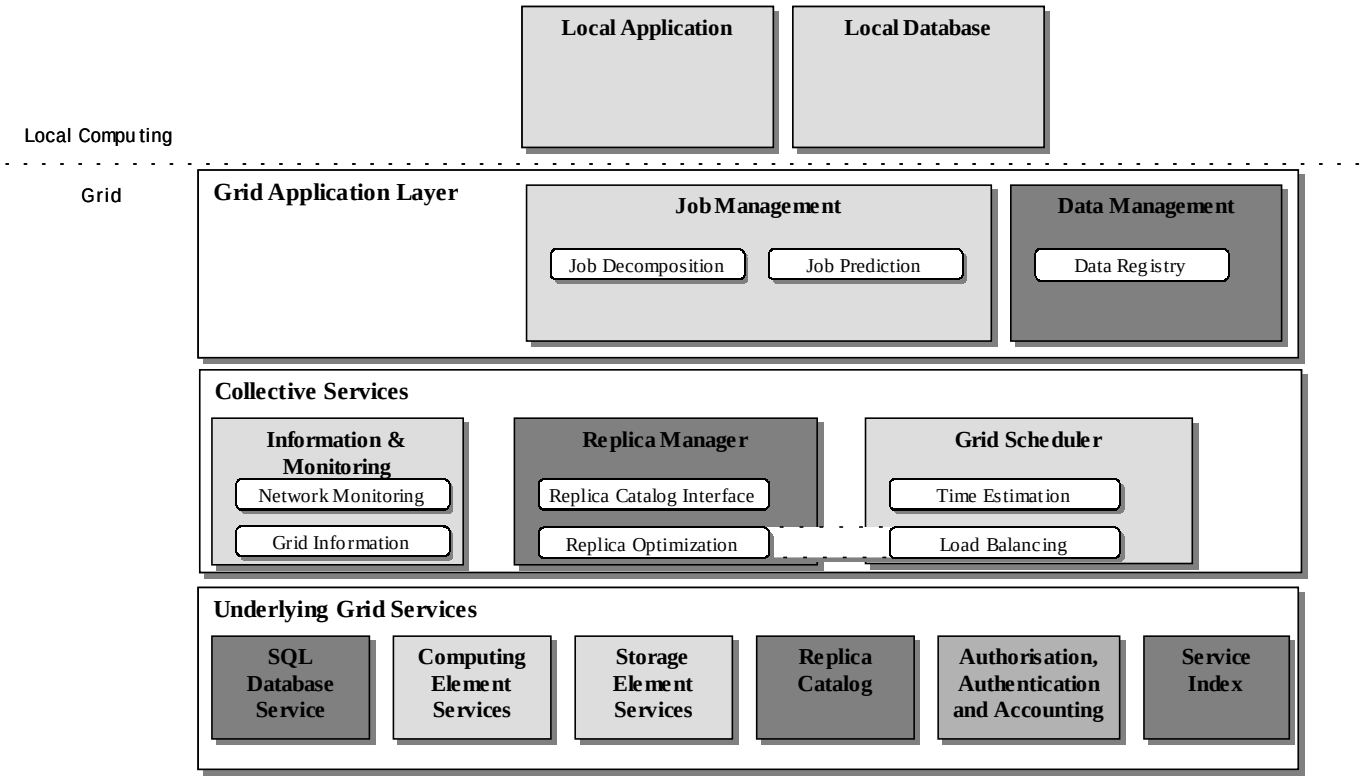

Figure 5.8: Grid Architecture from the point of view of “Grid Query Optimisation”

In this section our aim is not to define an explicit "Grid Query Optimisation Service" per se, but rather to discover which services will be required in order to optimise data access in a Grid environment. Our focus is on the use case of High Energy Physics Analysis as described in Chapter 2.

Please be aware that this section is the most speculative and most visionary of our design sections. We feel that it is essential to keep these considerations as an active part of our design to gain insights into what future Grid infrastructures might be able to provide. Obviously these are all long-term considerations which need the active collaboration of all other DataGrid work packages.

5.5.1. The Local Application Layer

The top layer of the architecture, the "Local Application Layer", exists outside of the Grid infrastructure. It is in this layer that TAG data analysis will most likely be performed. TAG data will be stored on locally owned storage devices, using a locally managed database system. The optimisation of TAG data analysis is also part of the task "Grid Query Optimisation", and will be achieved through the *Bitmap Indexing* of TAG data values [R12, R14]. Once the physicist has exhausted the local supply of information, he begins to analyse on lower level Grid managed data, by submitting a job to the Grid via the local Grid Client.

5.5.2. The Grid Application Layer

5.5.2.1. Job Management

This application level *Job Manager* will submit each job to the *Grid Scheduler* in the form of a well defined "Job Description Language" (JDL, see [R15]). There is need, therefore, for *Job Decomposition* functionality within the *Job Management* system, so as to reformulate jobs into (graphs of) atomic computation steps ready for submission to the Grid. Only the applications themselves can provide enough information to fulfill such a decomposition task. One architecture (not necessarily advocated here) would be to define a standard interface to a *Job Decomposition* service, which some/all Grid applications could implement, such that the *Grid Scheduler* could access decomposition services at execution time. Another approach would be to decompose all jobs completely before passing them in JDL form to the *Scheduler*. The main output of the *Job Management* would thus be a logically decomposed job which could then be further "optimised" by the Scheduler together with the Replica Manager/Optimiser. It is important to note at this point that a close collaboration of WP1, WP2 and WP8 is required for addressing this topic efficiently.

We also propose the need for a *Job Prediction* component within each Grid application. Such a component would be responsible for collecting statistics on the execution of Grid jobs, and for making predictions based on those statistics in terms of:

- Which logical files / data collections a (analysis) job is likely to require.
- How long a job is likely to run for.

The collection of statistics for the first prediction would require that database access to files/data collections is logged in some way, and that the log information is made available to the *Job Prediction* component. (This may be done either by the Grid intercepting calls to the database, or by the database itself logging its own activities.) Assuming the gathering of statistics is possible, why would such predictions be important, and why would the information not be known already? In fact, in many analysis type jobs, it is impossible to know a priori what data the job will access, or how long it is likely to run for, due to the so-called “navigational problem”. Estimations made by the *Job Prediction* unit would then be used by the *Scheduler* to optimise the execution of the job. Since the similarity between different jobs can only be assessed on a semantic level (i.e. by the application submitting the job), the *Job Prediction* unit needs to be co-located with the Grid application. The unit may also use other information, such as the physicist's user profile, or processing hints the physicists have given, to make predictions for job execution. As was the case for the *Job Decomposition* system, there are two mechanisms by which the prediction information can be passed from the Grid application to the *Scheduler*. The first option is to include such information as hints in the JDL (i.e. to generate all the prediction information a priori). The second option is to define an interface for the *Job Prediction* unit which would be used by the *Scheduler* to make predictions if and when required.

5.5.2.2. Data Management

In the current context of optimisation we propose to extend the Data Management module of the application layer described in Chapter 3 with a *Data Registry*. It is an interface in the application layer to the Replica Manager of the Collective Services layer.

5.5.3. The Collective Services Layer

We define a few submodules as part of our discussion of optimisation. See the differences between Figures 5.8 and 3.1 in the Collective Services layer. In the sections below the functionalities of these modules with respect to optimisation are discussed.

5.5.3.1. The Information and Monitoring Service

The *Information and Monitoring Service* will aggregate information provided by monitoring services such as the *Network Monitoring Service* (monitoring traffic on Virtual Private Network links between sites) and the *Fabric Monitoring Service* (monitoring the load on computational resources at the different sites).

5.5.3.2. The Replica Manager

The aim of the *Replica Optimisation* module is to automatically replicate data throughout the Grid in such a way as to minimise the total cost of data access for all the jobs executing on the Grid. This is part of what we called “long term optimisation”. Within a site the module might control which files remain staged on disk and which ones are relegated to tape storage. Between sites the module would control which files are replicated and which ones are not. The question then becomes, how does the *Replica Optimiser* decide which files to replicate and which ones not to? i.e. how does the *Replica Optimiser* know which files will be in demand on the Grid and which ones will not? It could do this by:

- collecting its own statistics; (It would be too late to collect information at this point, if the mapping between data collections and files is not constant across the Grid, i.e. if internal data reclustering at each site causes data to be stored differently at each site.)
- asking the *Grid Scheduler*
- accessing standard services of the application layer *Job Prediction* module or by
- receiving hints directly from the application. (The application level *Job Prediction* unit could assign importance levels to different logical files / data collections, which the *Replica Optimiser* would then to decide where and how many replicas to create.)

A second function of the *Replica Optimiser* (a kind of short term optimisation) is to find the "best" replica of a file when the file is demanded by the *Scheduler*. In this case the *Replica Optimiser* can decide whether it should create a new replica of the file locally, create a temporary copy of the file locally, or (possibly) open the file on the remote location for remote access. In the case where it decides to create a local replica or copy of the file, the *Replica Optimiser* must use an interface provided by the *Data Registry* module of the application level *Data Management* system to import the new replica/copy into the local database implementation.

5.5.3.3. The Grid Scheduler

Other important functions of the *Replica Optimiser* are to supply the *Time Estimator* module of the *Scheduler* with time estimates for the retrieval of files, and to negotiate with the *Load Balancer* to determine the best location for executing a given job, based on both the data and computational requirements of the job. The negotiation with the *Load Balancer* may require close coupling between the two components (as shown in Figure 5.8). We want to stress again at this point the importance of a close collaboration between WP1 and WP2 on this task.

The *Scheduler* provides high level scheduling services to Grid applications such that applications need not know where and how to schedule their work on the Grid, but can simply define the constraints for

running a job (such as the amount of memory it requires, the input data collection it needs, etc.) and allow the Grid to schedule it for them. The applications can then view the Grid as a single enormous computation and storage resource. One of the functionalities that needs to be provided by the *Scheduler* is that of *Time Estimation* for the execution of a Grid job. A time estimate is used by the Grid application or the physicist to decide whether or not to run a job. The *Time Estimator* uses information from the *Job Prediction* module, the *Information and Monitoring* services, and the *Replica Optimiser* to calculate an approximate time for job execution.

The *Load Balancing* unit is responsible for the actual scheduling of jobs to different sites on the Grid. It takes as input a decomposed job from the Grid application, and negotiates with the *Replica Optimiser* to discover the best location for job execution, based on the availability of both data and computational resources. (The *Replica Optimiser* uses information from the monitoring services to compare the costs for replicating data to different sites.)

5.5.4. Grid Query Optimisation (GQO) Task Specification

The GQO Task is foreseen to form a major part of the Replica Optimiser module of the Replica Manager. In this section we look more closely at the required functionality of such a module. The functionality can be viewed in terms of the five "services" it offers:

- **Data Access Time Estimation.** Aid the Time Estimation unit of the Scheduler to calculate approximate data access times for jobs that might be submitted to the Grid. The Replica Optimiser accesses the Network Monitoring Service (to discover the current network bandwidth situation), a Storage Device Monitoring Service (to discover the current load on the devices), and the Replica Catalog (to discover the amount of replication of the required files), so that an estimate of the cost of data movement can be returned to the Scheduler.

- **Pre-Execution Optimisation.** Aid the Load Balancing unit of the scheduler to make scheduling decisions, (i.e. help the Load Balancer decide on which site to run a job). An optimised scheduling decision should take into account both the cost of data movement between sites and the computational load on sites. A trade-off between data optimisation and computation optimisation will be achieved through the use of a negotiation protocol between the Replica Optimiser and the Load Balancer. (This negotiation/interaction protocol is still to be defined.)

- **During Execution Optimisation.** If a job requests a set of data at a particular site, and the data is unavailable at that site (e.g the database method returns an exception, and the exception is caught by the Replica Optimiser), then the Replica Optimiser needs to make a decision on how to optimally access the missing data. The Optimiser can then choose between possibly five options:

    - Open data for remote read on a site with a fast network connection.
    - Copy the data locally, register and create a replica of the data.
    - Copy the data locally, register a temporary replica and de-register it subsequently.
    - Ask the Scheduler to reschedule and restart the job on another site (e.g. a Tier 1 or 0 site).

- **Post-Execution Optimisation.** The aim of this optimisation is to automatically distribute (create replicas of) the Grid managed output files created by jobs running on the Grid. (Such

output files are primarily only created by production type jobs, which are not the main focus of our work). In making decisions on to what extent to replicate the output files, the Optimiser relies on hints from the user submitting the job, as well as heuristics information on the use of similar sets of data.

- **Offline Optimisation.** The aim here is to monitor the usage of logical files / data collections on the Grid as a whole and try to match the supply of replicas to the demand for them.

As well as creating the major part of the Replica Optimiser, the GQO task involves helping the applications groups (Work Packages 8 to 10) to create the Grid Application Layer services they require in order to make optimal use of the Grid. The services of interest include the Job Decomposition, Job Prediction, and Data Registering service. The assistance would be in terms of defining standard interfaces to these components and possibly helping to create generic code for use in each application's implementation of the components.

5.5.5. Interaction of Services for the HEP Use Case

We now return to the particular HEP use case described previously in Chapter 2.

At the start of the use case, the physicist performs "cuts" on the entire data set, by specifying that she is only interested in those events for which certain conditions on TAG attribute values hold. The local application sends these "cut predicates" to a local (bitmap) indexing system, which returns a list of events adhering to the selection. The physicist then performs some sort of statistical analysis on the events returned by the indexing system, studies the results and repeats the process. Since TAG data is mostly stored locally, the Grid is probably unaware of the analysis being performed by the physicist. After exhausting the information available in the TAG data, the user writes analysis code and submits it as a job to the Grid. The local application sends this code, along with the names of the AOD objects of interest and an output location for the results of the computation, to a Grid Application Layer "HEP Application".

The Grid level application reformulates the request into a job capable of execution on the Grid. It does this by mapping all of the requested AOD objects to a set of logical files or to a data collection description. It may also decompose the job into smaller subjobs via the Job Decomposition service. Having reformulated the request, the application then submits it to the Grid for an estimation of execution time and cost, by sending a request to the "Scheduler" which provides an interface to the services of the "Time Estimator". The Time Estimator requests information from the Job Prediction module (data and time requirements of the job), the Monitoring Services (current computational load on the Grid), and the Replica Optimiser (cost of data retrieval) to calculate an approximate time for job execution. When the application receives the execution time estimate, it uses that information to schedule the execution of the job on an application level, based on the cost of the job, the user submitting the job, the status of its job queue, etc. Of course the Grid application could also decide to refuse to schedule the job, or just to send the time/cost estimate back to the local application for approval. For more precise estimates, the Time Estimator could also contact the Load Balancer (see below).

In order to decide on which site to schedule the job, the Load Balancer enters into a negotiation process with the Replica Optimiser. The Load Balancer first uses the constraints given in the JDL job description (such as memory requirements, software library availability, etc.) to select a set of "possible sites" for job execution. It then requests information from the Fabric Monitoring service and the Job Prediction service in order to calculate the computation cost for job execution on each of these possible sites. It also sends the list of possible sites to the Replica Optimiser, so that the optimiser can

use information from the Network Monitoring service and the Replica Catalog to calculate the minimum cost of staging/using/creating the required data at each of the possible sites. The Replica Optimiser then provides this information to the Load Balancer, so that the Balancer can schedule the job at the site with the lowest overall cost. (The "negotiation" between the Load Balancer and the Replica Optimiser given above is highly simplified, and implies a full search of the available search space, to achieve a global minimum. Such an exhaustive search may not be possible or be simply inefficient, in which case more complicated forms of negotiation may be required. Further discussions with WP1 are required here.)

The Load Balancer then asks the Replica Optimiser to stage all required files to the site selected for execution. Once the files have been staged, the Load Balancer dispatches the job for execution to the selected site, using the Remote Job Execution service.

During job execution, the navigational access within the job causes it to demand a set of data which is not available in the local database. The Replica Optimiser catches the exception thrown by the database, and remedies the situation by either opening the data for WAN access on a remote site, copying the data locally to create a temporary/permanent replica, or asking the scheduler to stop the job and restart it (from the beginning) on another site. (The latter might be the case if a job on a Tier 2 site say, wants to start accessing Raw data, in which case restarting the job on a Tier 0 or 1 site, would probably be more advisable than moving Raw data to the Tier 2 site.) . Certainly the "navigational" case can also be handled by the Replica Manager. However, since the input data set is not known in advance, no explicit optimisation techniques concerning the optimal replica selection can be applied. Also a possible estimation about the access time for this job cannot be made. The only possibility for optimisation would be to deliver the requested files from a "cache".

Once the job has reached completion, the Replica Optimiser is responsible for deleting any temporary replicas created for the execution of the job. (Deletion also implies the deregistering of the data from the local database).

5.5.6. Simulating Grid Query Optimisation

A further aim of WP2 is to build a simulated DataGrid environment, and to attempt to optimise data access within such an environment. More specifically, this simulator has the following goals:

- To build a system that can realistically simulate a Grid environment, in which multiple autonomous resources must be managed coherently. The model will include simulations of the main parts of the architecture components discussed above. We will simulate a "simple" application requesting a set of logical files. We will also simulate major parts of the Replica Manager and the Grid Scheduler to study and optimise the complex "negotiation process" between these components. In particular, we plan to simulate optimal selection of replicas. Based on access patterns, a further goal is to study the impact of "automatically" creating data replicas between sites.

- The simulator will help testing different algorithms and heuristics for making such replication decisions, based on their ability for optimising globally the use of data resources (disk arrays and tape pools) on the Grid.

Additional goals in building the simulator include:

- To study the effects of local policy decisions on the overall working of the Grid system. For instance, what is the impact of reducing the disk quota for a certain user community? How shall a user community with a high priority be handled?

- To validate the design and the interfaces of the different Grid components.

A final goal is that the Replica Optimiser - initially a simulator only - is a software component which is part of the Replica Manager. This additional software component uses monitoring and performance information and optimises replica selection based on current performance parameters and predictions in the Data Grid.

It is beyond the scope of this document to describe the specifications of the simulator. Such specifications will be given in due course in another document. At this time a preliminary Grid environment simulator has already been built using Belief Desire Intention (BDI)-based software agents. A first prototype of this simulator was presented in [R3].

6. SERVICE API'S

6.1. INTRODUCTION

The following services described in this section will be provided by WP2. The detailed design of each service was described in Chapter 5.

The services are:

- GDMP
- FileCopier *
- ReplicaCatalog *
- ReplicaManager *
- SQLDatabase *
- ServiceIndex *
- ReplicaSelection
- ReplicaConsistency

We currently only have detailed API's worked out for the services marked with a *. Note that GDMP provides command line tools and no programming language API's.

6.2. GDMP

6.2.1. Interfacing GDMP

In contrast to other services in this document, GDMP does not provide programming language API's but command line tools.

6.3. FILECOPIER

The FileCopier Service is a low level service with a simple interface. (Note that an application should use the ReplicaManager API which in turn uses the FileCopier and updates ReplicaCatalogs).

6.3.1. API

- ***copyFile(TransportFileName source, TransportFileName destination):Status***

  This function also allows for third party transfer.

- ***setTransferParameters(timeout, parallel, striped, buffersize, restartOnFailure)***

6.3.2. Protocols

The GridFTP protocol will be used.

6.3.3. Services needed

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

6.3.4. Constraints and assumptions

None.

6.3.5. Open issues

None

6.4. REPLICACATALOG

6.4.1. Protocols and API's

In order to achieve maximum flexibility we decouple transport protocol, query mechanism, and database backend technology. This allows the implementation of a ReplicaCatalog server using multiple database technologies such as RDBMSs, LDAP-based databases, or flat files. Lean and narrow API's and protocols between client and server are required. This excludes the use of mechanisms specific to a particular database or query technology, such as SQL or LDAP query. We propose to use GSI-enabled HTTPS for transport, and XML for input/output data representation. Both HTTPS and XML are the most widely used industry standards for this type of problem. We are currently planning to use servlets to implement the HTTPS interface to the databases.

6.4.2. API

It will be the Replica Manager that uses the Replica Catalog API. However, experienced users or other Grid tools like the Grid Scheduler might also have direct access to the Replica Catalog. 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, file type 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)***

  Adds a logical file in the replica catalog.

- ***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***

6.4.3. Protocols

XML or URL encoding over https.

6.4.4. Services needed

Globus Replica Catalog (based on LDAP) and SQLDatabaseService.

6.4.5. Constraints and assumptions

None.

6.4.6. Open issues

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

6.5. REPLICAMANAGER

6.5.1. 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 selects 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 selects 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.

- ***getBestPhysicalFileName (PhysicalFileNameList, ProtocolList): PhysicalFileName***

  Returns the best physical file name and filter the output based on certain protocol. A list of possible PFNs is provided as an input parameter and the Replica Manager selects the best location based on performance information.

- ***getTransportFileName (PhysicalFileName, Protocol): TransportFileName***

  Returns the TFN for a given protocol.

- ***getPosixFileName (TranportFileName): filename***

  Returns a conventional POSIX file name that can be used to open a file using a standard POSIX open rather than Grid tools.

6.5.2. Protocols

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

6.5.3. 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

6.5.4. Constraints and assumptions

None.

6.5.5. 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 liberally: 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 revised.

6.6. SQLDATABASESERVICE

The SQLDatabase Service is described in Chapter 5.

6.6.1. 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 for load balancing
- Servlet: SQLServlet + Globus CoG
- SQL to XML mapper: XSQL et al
- Database driver and RDBMS: JDBC + Oracle, MySQL, DB2, etc.

6.6.2. API

- ***doHTTPGet(URL url)***

  Sends request with parameters indicated by URL to the SQLDatabaseService; returns HTTP response in XML format

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

  Sends request with parameters indicated by URL to the SQLDatabaseService, using given message body; returns HTTP response in XML format.

6.6.3. Protocols

https, X.509 (GSI compatible)

6.6.4. Services needed

None.

6.6.5. Open issues

None

6.7. SERVICEINDEX

The ServiceIndex is described in Chapter 5.

6.7.1. Class and Object Diagram

6.7.2. API

- ***registerChild(ServiceDescription)***

  Periodically used by a service to (re)announce itself as being a child of the service index.

- ***getChildren() : List of ServiceDescription***

  Returns all child service descriptions known to the service index

- ***getParents() : List of ServiceDescription***

  Returns all parent service descriptions known to the service index

6.7.3. Protocols

https, X.509 and user/password auth.

6.7.4. Services needed

SQLDatabaseService.

6.7.5. 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.

6.7.6. Open issues

None