Database Replication in World-wide Distributed Data Grids

eingereicht von:
Mag. Heinz Stockinger

Dissertation

zur Erlangung des akademischen Grades
Doctor rerum socialium oeconomicarumque
(Dr. rer. soc. oec.)
Doktor der Sozial- und Wirtschaftswissenschaften

Fakultät für Wirtschaftswissenschaften und Informatik
Universität Wien

Erstgutachter: ao.Univ.-Prof. Dr. Erich Schikuta
Zweitgutachter: Univ.-Prof. Dr. Dr. Gerald Quirchmayr
CERN Betreuer: Dr. Ian Willers

CERN (European Organization for Nuclear Research), Genf, Schweiz,
19. November 2001

Contents

1 Introduction **5**
- 1.1 Contribution of this Thesis . . . 6
- 1.2 Outline and Structure . . . 7

2 Data Replication **9**
- 2.1 General Introduction to Replication . . . 9
- 2.2 Transaction Theory and Concurrency Control . . . 12
- 2.3 Synchronous and Asynchronous Replication . . . 13
- 2.4 Categorisation of Replication Approaches . . . 14
  - 2.4.1 Update Mechanisms . . . 15
  - 2.4.2 Voting and Quorum Mechanisms . . . 16
- 2.5 State-of-the-Art in Replication . . . 18
  - 2.5.1 Discussion on Synchronous versus Asychnronous Replication . . . 18
  - 2.5.2 Lazy Update Propagation Approaches . . . 19
  - 2.5.3 Group Communication Approaches . . . 20
- 2.6 Commercial Database Management Systems . . . 21
  - 2.6.1 Objectivity/DB . . . 21
  - 2.6.2 ObjectStore . . . 23
  - 2.6.3 Versant . . . 23
  - 2.6.4 Oracle . . . 24
  - 2.6.5 Sybase . . . 24
  - 2.6.6 DB2 . . . 24
  - 2.6.7 Mircosoft SQL Server . . . 24

3 Data Grids and Data Management Challenges **25**
- 3.1 The Vision of a Grid . . . 25
- 3.2 Data Management Problems . . . 26
- 3.3 Data Management Challenges in High Energy Physics . . . 26
  - 3.3.1 CERN and High Energy Physics . . . 27
  - 3.3.2 The CMS Experiment and the Data Model . . . 28
  - 3.3.3 Data Challenges: Distribution and Replication . . . 31
- 3.4 The European DataGrid Project at CERN . . . 33
  - 3.4.1 Aim and Background . . . 33
  - 3.4.2 Work Package Categorisation . . . 33
  - 3.4.3 Related Data Grids in the High Energy Physics community . . . 34
- 3.5 File and Object Replication . . . 35
  - 3.5.1 Motivation for Different Replication Granularities in HEP . . . 35
  - 3.5.2 Distributed Object Systems . . . 36
  - 3.5.3 Distributed File Systems . . . 36
  - 3.5.4 Replication and Caching in the Internet . . . 37

3.5.5 Discussion . . . 37

4 Architecture and Cost Model **40**
4.1 Data Management Architecture . . . 40
4.1.1 Grid Services for Replication . . . 41
4.1.2 Replica Catalogue Design . . . 43
4.2 A Cost Model for Data Distribution and Replication . . . 46
4.2.1 Model Assumptions . . . 46
4.2.2 Location of the Data Store . . . 47
4.2.3 Costs and Influences of the Data Store . . . 47
4.2.4 The Cost Model . . . 50
4.2.5 Related Work on Cost Models . . . 52
4.3 Example . . . 52
4.3.1 Introduction . . . 52
4.3.2 The Minimisation Problem . . . 53
4.3.3 The Logical OID and the Job Model . . . 53
4.3.4 Architecture of the Object Location Table . . . 54
4.3.5 Use of the Object Location Table in Sub-job Execution . . . 55
4.3.6 Applying the Cost Model . . . 56
4.3.7 Concluding Remarks . . . 57

5 Grid Data Mirroring Package (GDMP) **58**
5.1 Introduction . . . 58
5.2 Requirements and Basic Functionality . . . 59
5.2.1 Distributed Data Production in CMS . . . 59
5.2.2 A Basic Usage Scenario . . . 59
5.2.3 The File Replication Process . . . 60
5.3 Architecture . . . 61
5.3.1 Control Communication . . . 61
5.3.2 Request Manager . . . 62
5.3.3 Security Layer . . . 62
5.3.4 Database Manager . . . 62
5.3.5 Data Mover . . . 63
5.3.6 Replica Catalogue . . . 63
5.3.7 Storage Manager . . . 64
5.4 The Data Replication Process and Policies . . . 64
5.4.1 File Catalogues . . . 64
5.4.2 Subscription and Replication Model . . . 66
5.4.3 Notification System . . . 66
5.4.4 System States for File Replication Process . . . 66
5.4.5 Partial Replication: Filtering Files . . . 67
5.4.6 Fault Tolerance and Failure Recovery . . . 68
5.5 GDMP Applications and Interfaces . . . 68
5.5.1 GDMP Server . . . 69
5.5.2 GDMP Client Applications . . . 69
5.6 Performance Considerations and Results . . . 70
5.7 Conclusion and Future Work . . . 71

6 Distributed Replica Catalogues **73**
6.1 Introduction . . . 73
6.2 Replica Catalogues and Related Work . . . 74
6.2.1 Related Work in General . . . 74
6.2.2 The Globus Replica Catalogue . . . 74
6.2.3 Motivation for a Distributed Replica Catalogue . . . 75
6.3 Architecture of a Distributed Catalogue Approach . . . 75
6.3.1 Catalogue Distribution . . . 75
6.3.2 HTTP for Interaction with the Catalogue Server . . . 77
6.3.3 HTTP Client-Server Interaction . . . 78
6.3.4 Grid Lookup Protocol - GLP . . . 79
6.4 Implementation Details and General Remarks . . . 79
6.4.1 Server-side Caching . . . 80
6.4.2 Logical Filenames . . . 80
6.4.3 Redirection of File Transfer Requests . . . 81
6.5 Catalogue Re-synchronisation . . . 81
6.5.1 The Stale Data Problem . . . 81
6.5.2 Re-synchronisation Implementation . . . 82
6.6 Experimental Results . . . 82
6.6.1 Cache Performance . . . 83
6.6.2 Wide-area Lookup Performance . . . 84
6.7 Conclusion and Future Work . . . 84

7 Replica Update Synchronisation **86**
7.1 Update Issues for Replicated Files . . . 86
7.1.1 A Motivating Example . . . 86
7.1.2 Infrastructure Required for Replication Middleware . . . 87
7.1.3 Methods for File Replica Updates . . . 88
7.2 Consistency Service . . . 91
7.2.1 Consistency Requirements . . . 92
7.2.2 Levels of Consistency . . . 93
7.2.3 Architecture . . . 93
7.3 Replica Update Propagation Protocols . . . 95
7.3.1 File Replacement Protocol . . . 96
7.3.2 Binary Difference Protocol . . . 98
7.3.3 Log-Based Protocol . . . 100
7.4 Protocol Discussion . . . 102
7.4.1 Lock Contention and Local Updates . . . 103
7.4.2 Comparison of the Three Protocols . . . 103
7.4.3 Concurrency Control and Admission Control . . . 105
7.5 Conclusion and Future Work . . . 105

8 Access Optimisation for Replicated Data **106**
8.1 Introduction . . . 106
8.1.1 A Motivating Example . . . 106
8.1.2 Infrastructure and Architecture . . . 107
8.1.3 Related Work . . . 108
8.2 Performance Parameters . . . 109
8.2.1 Network . . . 109
8.2.2 Data Server . . . 109

8.3 Replica Selection 110
8.3.1 A Cost Model for Replica Selection 110
8.3.2 Replica Selection and Data Consistency 112
8.3.3 Discussion 112
8.4 Advanced Issues 112
8.4.1 Implications for Grid Applications 113
8.4.2 Dynamic Replication 113
8.5 Conclusion and Future Work 114

9 Conclusion **115**

Chapter 1

Introduction

In distributed systems, data replication is a well known and accepted technique for optimising data access and providing fault tolerance. This is achieved by storing multiple copies of data at several locations. In particular, distributed database management systems, distributed file systems, distributed object systems as well as Grids deal with replication aspects but tackle the problem from different points of view. The topology and the latency of the network have an important influence on the replication strategy used. We analyse replication issues for wide-area networks with a large user community and huge data stores distributed over several sites on the globe. Our application domain is mainly the scientific community and in particular High Energy Physics.

In this thesis, we focus on replication strategies and problems in database management systems and Grids, in particular Data Grids. Commercial as well as open source database management systems provide replication features but they mainly do not satisfy the requirements of world-wide distributed user communities for efficiently and securely replicating large amounts of data in the Petabyte scale between several storage locations on the globe. This is true despite the fact that the replication problem is rather old and database researchers have been working on solutions for more than two decades. On the other hand, compared to the long existence of the database community, the Grid community is rather new and has just recently started to tackle data replication problems. Resource distribution over wide-area networks and use of some Internet technologies are significant for Grid based software solutions.

In a Data Grid, computing and storage resources are distributed to several sites at different locations connected via wide-area network links. One of the major challenges is the data management aspect of large, distributed and replicated data stores. Recent Data Grid projects like the European DataGrid Project at CERN (the European Organization for Nuclear Research) have been established to address data management problems as well as more traditional job scheduling problems.

In a typical Data Grid, database files or simply plain files of arbitrary file type are to be replicated to several sites. File transfer mechanisms like FTP and established security infrastructures like Public Key Infrastructure (PKI) provide a basis for efficient and secure file replication. Currently, Grid technologies offer promising software solutions for read-only file replication. However, replica management not only deals with data transfer but also with meta-data management like replica catalogues and consistency management for file and meta-data updates. In the Grid community, file replica catalogues are currently under research but consistency models are still not sufficiently dealt with. In contrast, several approaches to replica consistency management and synchronisation are proposed by database researchers as well as commercial and open source solutions.

1.1 Contribution of this Thesis

This thesis has been carried out at CERN within the European DataGrid Project and in particular in the CMS experiment. CMS is participating in the DataGrid project since it is currently using Grid prototype tools and will base its distributed computing model more and more on Data Grid technology. Starting from 2006, large amounts of data (in the order of a few Petabytes) will be recorded by particle detectors and stored in object stores afterwards. Since production runs of simulated data, which include replication of files between several sites in Europe and in the United States, are already done now (i.e. in the year 2001), first Grid prototypes needed to be designed, evaluated and tested before actually being used in a production environment of the running experiment.

The requirement of CMS for prototyping replication tools as well as the initiation of the European DataGrid projects are driving forces for this thesis. Since software projects and in particular the Grid projects are collaborations of several people, parts of the contributions of this thesis have been achieved in co-operation with colleagues and is pointed out explicitly.

This thesis has several contributions to the Grid research community in theoretical and practical aspects:

- Data Grids are rather new and only preliminary solutions have been proposed so far. The Grid Data Mirroring Package (GDMP) project has been initiated and we have proven with architecture and a software solution that Data Grid tools can be used in real, production Data Grid environments for secure and efficient file replication. The software project is in principle a mutual project with Asad Samar (California Institute of Technology). Pioneer steps include the evaluation of existing tools (in particular the Globus Toolkit), architectural design of a data replication software package and finally implementing the software. Thus, we have shown that parts of the Data Grid vision are already realised today. Besides research projects, in Grid computing lots of visionary publications and projects have been created. However, in this thesis we provide theoretically and practically sound solutions for file replication.

  In later stages of the GDMP project, a few more people have contributed to the software.

- In the European DataGrid project, data management is a major task. Whereas data management includes several sub-tasks for replication, meta-data management, query optimisation and security, the GDMP project has a well defined focus on replication for read-only files. Within the data management team we contributed largely to architectural design of data management solutions.

- Another main contribution of this thesis is to identify data management problems in a Data Grid - in particular data replication issues - and provide solutions which are partly known in the database community but not common in the Grid community. Up to now, the distributed database community and the Data Grid community have been rather independent communities. We provide a first contribution to find commonalities in both communities and bring both worlds together in order to have an efficient Data Grid. We mainly focus on object-oriented databases.

- File replication tools require catalogues for storing information about file names and locations. A particular Grid replica catalogue technology in the Globus Toolkit is mainly based on a central catalogue. We propose a scalable and easy manageable distributed replica catalogue based on HTTP redirection.

- Since Data Grids do not only use read-only data but some data needs to be updated and synchronised, we propose theoretical and practical approaches to deal with replica

synchronisation and consistency of files in a Data Grid. Furthermore, access optimisation to replicated files is analysed and discussed in detail.

1.2 Outline and Structure

We first give a chapter on formalisation and definitions of replication aspects. **Chapter 2 Data Replication** introduces replication techniques and gives the theoretical background. Most advanced replication models and approaches are proposed in distributed databases and database theory in general. Definitions provided in Chapter 2 are used throughout the entire thesis and are of major importance for the design of Grid replication strategies and tools. A short categorisation of replication protocols is given in order to provide a basis for the following state-of-the-art discussion on distributed database update synchronisation.

Chapter 3 Data Grids and Data Management Challenges provides a problem specification of Data Grids and sets the frame for the application domain and scope of this thesis. Based on the vision of the Grid, Grids and in particular Data Grid motivation and their problem domain are illustrated. In Grids, several data management problems are identified. The thesis has been written in the CERN environment which needs further explanation. In more detail, CERN is involved in High Energy Physics and Data Grid projects with main emphasis on High Energy Physics. Consequently, data management challenges of this particular domain are introduced.

In Chapter 3 we also provide a general overview of the European DataGrid project initiated and managed by CERN and briefly describe related Data Grid projects. Since several replication granularities are possible, e.g. file versus object replication, we provide a motivation for different replication granularities and then discuss shortly replication approaches in several distributed systems. A short comparison and the relevance to Data Grids is given.

Starting with **Chapter 4 Architecture and Cost Model**, we discuss our contributions and proposals to the research community. Each of the following 5 chapters (including Chapter 4) contains solutions and their justifications. Examples are mainly based on the Data Grid requirements. The order of the chapters is selected in a way that Chapter 4 provides an architectural solution to general data management and replication problems, and gives a detailed cost model for replicated data stores whereas Chapters 5, 6, 7 and 8 provide solutions to very particular sub-sets of the architectural components pointed out in Chapter 4. The last section of Chapter 4 reports on early work on an object location table for locating and efficiently accessing objects distributed and replicated to large object stores. This is the used for an example of the cost model.

Chapter 5 Grid Data Mirroring Package (GDMP) discusses details of the architectural design and functionality of the GDMP software and gives some performance results for file replication.

Originally, GDMP did not use a replica catalogue for managing file location information. However, in a later version the Globus replica catalogue has been introduced as a component of the software package. The Globus replica catalogue is currently based on a central replica catalogue and some distribution based on LDAP referrals is possible. In **Chapter 6 Distributed Replica Catalogues** we propose a distributed replica catalogue system based on HTTP redirection. A replica catalogue client first connects to a central replica catalogue and is redirected to a site catalogue which then resolves the actual physical file location.

Chapters 4 to 6 propose file replication solutions for read-only files. In contrast, **Chapter 7 Replica Update Synchronisation** deals with updates of replicated files and possible consistency models. Since replica update synchronisation has different limitations and constraints in a Data Grid than in a distributed database management system, Grid update synchronisation is motivated and theoretically discussed.

One main goal of replication is to optimise access to data and bring data close to the user. **Chapter 8 Access Optimisation for Replicated Data** discusses optimised access to replicated data and replica selection based on performance information.

Finally, concluding remarks and a short outlook to future work on replication within the DataGrid project is given.

Chapter 2

Data Replication

This chapter provides a general introduction, formalisation and state-of-the art to data replication. Data replication is used in several distributed systems like distributed database management systems, distributed objects systems (e.g. CORBA), distributed file systems and Data Grids (see Chapter 3 for background on Data Grids). Each of the mentioned distributed systems provides different preconditions and environment specific assumptions and restrictions. For more details on system specific parameters and problem specifications refer to Section 3.5.

Here, we provide a general overview of replication issues, define the most important terms, thus the environment for the thesis, and then concentrate on replica updates and synchronisation. Since distributed database management systems provide the most complete mechanism for replica updates, emphasis is first put on database management systems and later in this thesis on some more general file and object replication mechanisms. Note that the main focus of the thesis is on data replication issues in Data Grids but we first start with most complete and general replication aspects and then discuss which of the general issues are applicable to the Data Grid environment.

2.1 General Introduction to Replication

Data storage and collections of data can be treated in a variety of ways. In the computing history, several storage technologies and paradigms have been introduced and are widely used today. Starting with single storage locations like the main memory of a machine, several ways of distributing data and data stores are known.

An important factor for storage devices and storage technologies is access time to the data. Based on different hardware storage technologies like cache memory, main memory, disk storage (secondary storage) or even tape storage (tertiary storage), the access times vary significantly. Here, we only consider secondary and tertiary storage. We assume that storage hardware is distributed and accessible via a local-area network (LAN) or a wide-area network (WAN).

LANs and WANs do not only allow for data distribution to several storage locations in the network but also allow for distribution of users and thus client applications. Although network latencies will decrease in the future since network technology improves, there is still a performance difference between accessing data locally (on the same machine) or remotely over a network. By providing a copy of a data item (see definition below) close to a client application, access times can be reduced. In general, managing copies of data items is regarded as *replication*.

Definition: A *data item* can be any unit of digital storage: it can be a single bit, a few bytes, objects or even an entire file.

In the general introduction to replication we use the word *data item* and thus do not limit the discussion to any specific unit. In Section 3.5 we discuss this topic in more detail.

Data replication is not only used for gaining performance in access times and hence hiding access latencies but also for dealing with problems that occur in distributed systems. The probability of failures in a distributed system is higher than in a central system with a single storage device. Networks can be partitioned and consequently remote resources are not available. In general, data replication can be applied for providing good performance, high availability and fault tolerance.

- **Performance and data locality**: When data is stored at a single location or a single data server, this server can be a bottleneck if too many requests need to be served at the same time and the whole system slows down, i.e. slow response time and limited throughput capacity in terms of requests per second. By offering multiple replicas at multiple locations, requests can be served in parallel and one replica provides data access to a smaller community of users. Note that replication speeds up response time for read requests but can reduce write requests due to protocol overheads. See Section 2.5.

  If multiple users access data over a network from geographically far distances, data access will be much slower than in a small local-area network given that LANs have lower network latencies than WANs. By providing data as close to the user as possible (data locality), smaller distances over the network also contribute to higher performance and lower response times.

- **Availability and fault tolerance**: If a single data item is only stored at a single server, this data item cannot be accessed if the server crashes or does not respond. On the other hand, if a copy (replica) of this data item is stored at a different server, this additional server can provide the data item in case of a server or network failure. Thus, the availability of data can be increased.

  Fault tolerance might also be considered as a kind of security that data will still be available after technical problems like machines crashes, hardware failures or even natural disasters like earthquakes.

In addition to these two main technical reasons, political and economical aspects (see also Section 4.2) can be stated. In particular, storage providers might want to have replicas in order to get higher hit rates on their data and might charge users more.

A simple way to look at replicas is to regard them as independent copies. For instance, one person creates a file and sends a copy to another person who can then use the private copy. Since the location of the second copy is neither stored in a file catalogue nor any guarantee is given that both copies have the same contents after one is changed, we do not regard these files as replicas.

As opposed to "conventional" data items that exist only once in a data store, replicated data items require a particular naming convention. A set of identical replicas is identified by a *logical name* and each individual replica is identified by a *physical name*. Let us assume a physical file called file1.DB which is stored at site X in the directory /data. Now an identical replica for file1.DB is created at site Y in a similar directory. Thus, the following two physical names exist: X/data/file1.DB and Y/data/file1.DB. A client application does not need to know about all physical names and thus about all physical locations of the file. For the client application it is sufficient to know the logical name, for instance file1.DB, and have an additional data structure that maps the logical name to a set of physical names.

Definition: A *logical name* uniquely identifies a set of identical replicas uniquely identified by physical names.

Definition: A *physical name* uniquely identifies a single replica at a single storage location.

Definition: A *replica catalogue* is a data structure or data storage that provides a mapping of logical to physical names, stores logical and physical names and keeps track of physical locations of data items (replicas).

Definition: An *independent copy* of a data item is neither registered in a replica catalogue nor any consistency between the original and the secondary copy of the same data item is maintained. Consequently, an independent copy can be changed without propagating the change to other copies.

Definition: A *primary copy* of a data item is the original data item that has been created first. All other replicas of the primary copy are called secondary copies. A primary copy is often called "master copy" or simply "master".

Thus, a replica is more sophisticated than an independent copy since it requires some data management and special data structures. A replica is dependent on at least one other original data item (primary copy). A simple view for replicas is when they are read-only and all the possible replicas always have the same content. The primary copy is registered in a replica catalogue and gets a logical name. A physical name identifies this particular instance at a particular location. When another replica (a secondary copy) is created, it is assigned the same logical name which gives a reference to the primary copy. Furthermore, an additional physical name is assigned.

Definition: A *replica* is a copy of a data item, is registered in a replica catalogue and might have a reference to other replicas for update synchronisation if the replica is updatable.

For a client application, the creation of replicas and their locations should be transparent. Let us illustrate this with an example. We consider a company that has two branches at two continents: one in Geneva, Switzerland, and another in Palo Alto, California (see Figure 2.1). Employees of the company want to have access to the same data. If data is only stored in Palo Alto, employees in Geneva need to wait longer to access data due to the latency of the wide-area network connection between Palo Alto and Geneva. Furthermore, due to current blackouts and power shortages in the Silicon Valley, the data might not be accessible at all to people in Geneva. In this case it makes sense to replicate data to both branches. An employee does not need to know about the replication process but only wants to access a physical instance of e.g. file1. If file1 is replicated, there are several possibilities to get the requested file. Replica access optimisation (see Chapter 8) discusses this topic in detail. Thus, the fact that replicas exist is transparent to the user. The end user only uses the logical name - in our example it is file1 - and the replication software will use the replica catalogue to resolve the logical filename to a physical filename and provide a replica which is optimal to the user in terms of access time.

Definition: *Transparency* in a replicated system refers to location and access transparency which means that data can be accessed by end-user applications without the exact location information.

Example: Replication tasks, advantages and disadvantages

We use the practical and real world example from above and explain the tasks, advantages and disadvantages of replication. Above we assumed that data is already in place and the two companies have access to their data. Let us now assume that the branch in Palo Alto

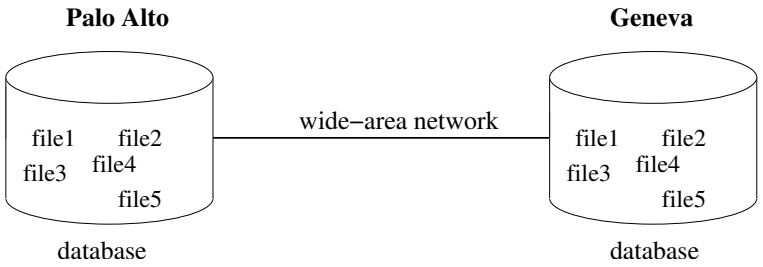


Figure 2.1: A company with two branches and a database replicated to both branches.

is writing new data (e.g. 1 GB of new costumer data) into the data store. Palo Alto may decide to write all information first locally and then notify Geneva that new data is available and ready for being transferred (replicated) to Geneva. Geneva can then decide if and when it wants to replicate new data.

Alternatively, Palo Alto may decide that the new costumer data is so important that Geneva needs to know immediately about the new costumer information. Thus, a synchronisation method is used to update both data stores at the same time. The disadvantage is that if the transatlantic network link is down, Palo Alto might not even be able to write into its own data store since only if both data stores are available, data can be written. However, Palo Alto may also choose a replication strategy that allows to write data locally and automatically updates the data store in Geneva when the network connection is back again.

Once data is replicated to Geneva and locally available to the European employees, the performance gain of data access is visible to the Geneva branch when accessing the new costumer data. The replication system uses internal data structures (replica catalogues) and performance information about the wide-area network connection and possible the data servers in Palo Alto and Geneva and then decides where to get the requested data from. Consequently, the employees in Geneva do not have to wait long when they want to display new data on their screens.

However, a disadvantage of replication as regards performance is that if changes to existing data are made, e.g. the name of a costumer is corrected, both data stores need to be synchronised which delays the update process.

2.2 Transaction Theory and Concurrency Control

In a simple replication environment, data items are ready-only and no updates can be applied to them. However, once data items need to be changed or updated, special care has to be taken in order to provide a consistent view on data when multiple, concurrent users request the same data. The update problem is not only an issue in distributed systems, but also in non-distributed systems like database management systems. We now discuss general database transaction issues for efficiently dealing with concurrent access to data and then elaborate on basic update problems in distributed and replicated systems. Specific solutions for replica update synchronisations will be proposed in Chapter 7.

Definition: A *transaction* is an atomic unit of database access which is either completely executed or not executed at all [21]. A transaction normally consists of several read and write *operations*.

Transactions are used for two reasons: first, for dealing with concurrent access to data and second for error recovery. Here, we are mainly interested in concurrent data access and dealing

with possible conflicts of concurrent read and write operations on the same data item. The following example illustrates a database transaction which has read operations on data items a and c and a write operation on data item b:

transaction = {read(a), write(b), read(c)}

A transaction is atomic when the entire sequence of read and write operations is executed and it does not have to interfere with other concurrent transactions. Conflicts between concurrent transactions occur when read and write operations are executed on the same data item. Consequently, concurrent transactions have to be serialised in order to prevent conflicting operations.

Definition: Concurrency control uses *serialisability* in order to guarantee that the result of concurrent transactions is the same as if they are executed serially in some order.

A basic method for serialising transactions is using locks. There are several possible ways to lock transactions but the most widely used locking mechanism is *2-Phase Locking (2PL)* [73]. It ensures consistency but at the price of a performance overhead. Briefly, the first phase is called the "growing phase" where new locks are acquired. In the second phase - the "shrinking phase" - the locks are released again. Consequently, the presence of locks prevents other transactions from reading or writing data items. Later in this chapter we discuss the negative aspects one faces when using 2-Phase Locking for distributed transactions on replicated data.

In a single, central database management system, a single server is responsible for concurrency control and serialisability. In a distributed database management system (or distributed system in general) where data items are distributed and replicated among several servers, each of them is responsible for a particular site and the local data items at this site. For instance, a client transaction updates a data item at site A and then the server at site B has to propagate the transaction to a possible site B.

Like in a non-distributed transaction, a distributed transaction can either atomically commit or abort. The most common commit protocol in distributed systems is the *2-Phase Commit Protocol* [73]. The protocol allows several servers to communicate with each other and come to an agreement if a transaction is committed or aborted. Briefly, in the first phase of the protocol, each server votes for the transaction to be committed or aborted and in the second phase each server carries out the joint decision. Often, 2-Phase Commit and 2-Phase Locking are used together to deal with concurrency control in distributed systems.

Another two essential features regarding consistency are *one-copy-equivalence* and mutual consistency. *One-copy-serialisability* is another expression for keeping replicas transparent, i.e. objects must appear as a single object. In other words, multiple copies of a data item must appear as a single logical item. The replication mechanism and its underlying distributed transaction system will enforce this. The correctness criterion is one-copy-serialisability that ensures both one-copy equivalence and a serialisable execution of transactions [3]. Note that the serialisable execution of transactions is a commonly accepted correctness criterion in databases and is enforced locally by strict 2-Phase Locking.

2.3 Synchronous and Asynchronous Replication

In a general distributed system there are many cases where single replicas get updated. This requires update synchronisation protocols that guarantee a consistent state when an application reads or writes data. In database research, update synchronisation of replicas is one of the most important points and in fact is a major efficiency and performance parameter for a replicated system.

There exist two main approaches for update synchronisation: *synchronous* and *asynchronous replication*. We explain both approaches using database terminology.

If data items are replicated, a distributed transaction has to operate on multiple identical data items. Instead of locking a single data item, all replicas of the same data item have to be locked when a write operation is done.

Definition: *Synchronous replication* requires that update operations on all replicas of the same data item are done within a single transaction. An equivalent term for synchronous replication is *eager replication* [49].

A synchronous replication approach requires that all or a majority of replicas are locked during a transaction. Thus, a write operation is executed on all replicas at the same time and replicas are synchronised within a transaction. When the transaction commits, all replicas are in the same state and have the same data values. Thus, a transaction can only complete successfully if several replicas acknowledge the change (update) successfully. Furthermore, all replicas are equal and every replica can be updated since all others need to be locked anyway. Consequently, all replicas are identical as regards data access control. In contrast, one might identify a master which is the only replica that can be updated by client applications and a server needs to propagate update information to other replicas.

A commonly used protocol for guaranteeing the correct order of operations within a transaction is the 2-Phase Commit Protocol. A disadvantage of this protocol is that it requires many messages to be sent over the network and many (or all) replicas have to be available at the time of the transaction execution (see Section 2.5.1). Voting and quorum techniques (see next section) are used for deciding if a transaction can be executed.

Definition: An *asynchronous replication* approach updates only one replica in a transaction and propagates the changes to other replicas later in an additional transaction. An equivalent term for asynchronous replication is *lazy replication* [49].

An alternative approach is to use asynchronous replication that updates only one replica during a transaction and propagates the changes to other replicas after the update (write) transaction has finished successfully on one replica. This approach has lower response times for write requests but data may be out of sync for some time. Manual or automatic reconciliation processes are used to converge to a common value of replicas.

Definition: *Replica consistency* assures that two or more replicas have the same data value. Replica synchronisation is used for establishing consistency.

Strict consistency can only be gained with synchronous replication. Since this strictness is not required by all applications, we can relax the consistency requirement by using asynchronous replication methods. Based on the relaxation of consistency, we can identify several consistency levels (see Chapter 7).

A detailed comparison of synchronous versus asychnronous replication can be found in Section 2.5 where we look at current state-of-the art in database replication issues.

2.4 Categorisation of Replication Approaches

Replication techniques in the literature as well as in commercial and open source database management systems use several different approaches to solve the data replication problem. Often, the replication approach is very much dependent on application requirements and on

influences and constraints like network topology, latency, number of update transactions, number of replicas - just to name a few. We provide a simple categorisation of the most often used approaches and use it later in this thesis to asses the relevance of a particular replication technique for a Data Grid.

The main context of this thesis is replication over wide-area network connections and thus the most dominant categorisation aspects are if a replication protocol uses synchronous or asynchronous replication methods and if the protocol is optimised for wide-area networks.

In [28] two different replication approaches are distinguished: the gossip approach and the process group communication approach.

Definition: In a *gossip approach* updates are propagated lazily to all the participating replicas, one after the other. In other words, single messages are sent to all sites where replicas reside.

Definition: In a *group communication approach* updates are multicast/broadcast to all replicas at the same time.

In practice, group communication approaches use either hardware or software broadcast operations. Whereas broadcast is rather easy on local-area networks, it is quite difficult to achieve over wide-area network connections since routers need to be configured to allow for IP multicast. Consequently, multicast is quite restricted to private networks where network administrators and thus replication systems have control over the network traffic.

For a more detailed categorisation on eager replication techniques with emphasis on group communication refer to [130].

2.4.1 Update Mechanisms

As regards client permissions and timing for replica updates, the following update mechanisms can be distinguished:

- *update anywhere*: This is the most general update replication mechanism where a client can update any replica in a distributed database, i.e. all replicas are equal and no masters are assigned. In contrast, the mechanism needs most strict update and locking mechanisms.

- *Primary-copy approach*: This is also known as "master-slave" approach. The basic idea is that for each data item or file a primary copy exists and all the other replicas are secondary copies [16]. Updates can only be done by a primary copy which is the owner of the data. If a write request is sent to a secondary copy, the request is passed on to the primary copy which does the updates and propagates the changes to all secondary copies.

  This policy is implemented in Versant [127] and ObjectStore [88] (see Section 2.6). Also Oracle [92] provides such a feature. The primary-copy approach provides a high degree of data consistency and has improved write performance features compared to synchronous replication because locks on files do not have to be agreed among all replicas but only by the primary copy.

- *Epidemic approach*: User operations are performed on any single replica and a separate activity compares version information (e.g. time-stamps) of different replicas and propagates updates to older replicas in a lazy manner (as opposed by an eager, synchronous approach) [2]. Thus, update operations are always executed locally first and then the sites communicate to exchange update information. The degree of consistency can be very low

here and this solution does not exclude dirty reads or other possible database anomalies. Such a system can only be applied for non time-critical data since the propagation of updates can also result in conflicts which have to be solved manually.

- *Subscription and relatively independent sites*: Similar to the epidemic approach, another policy is that a site only wants to have data and does not care about replica consistency at all. When any other site does updates on a particular file, only certain sites are notified of the updates. This follows a subscription model where a site subscribes explicitly to a data producing site. A site that has not subscribed is itself responsible to get the latest information from other sites. This allows a site to do local changes without the agreement of other sites. However, such a site has to be aware that the local data is not always up-to-date. A valid solution to this problem is to provide an export buffer, where the newest information of a site is stored, and an import buffer, where a local site stores all the information that needs to be imported from a remote site. For instance, a local site has finished writing 10 different files and puts the file names into the export buffer. A remote site can then be notified and it transfers the information from the export buffer to its local import buffer and requests the files if necessary. This approach allows for flexibility concerning data consistency and independence for a local site. A site can decide itself which data to import and which information to filter. Furthermore, a data production site may not export all the locally available information and can filter the export buffer accordingly.

  A valid implementation of this approach can be found in the Grid Data Mirroring Package (GDMP) (see Chapter 5). Web page mirroring where updates are propagated only once a day is another example for asynchronous replication.

- *ROWA (read one, write all)*: as the name suggests, for write operations, all replicas need to be synchronised and thus are written at the same time. Since all replicas are supposed to be in sync all the time, only one of the replicas needs to be read to have a consistent read operation. This mechanism corresponds to synchronous replication.

2.4.2 Voting and Quorum Mechanisms

Distributed systems have a higher probability for failures than centralised systems just based on the fact that several hardware and software resources are involved that might crash. A major criterion of a replication protocol is that it is still working in case of failures or unavailability of certain data items. Voting techniques and quorum mechanisms have been proposed for dealing with read or write operations in case some replicas of a data item are currently not available.

Voting methods have a long tradition and go back to 1979 where Gifford explained weighted voting for replicated data in [47]. Here, write and read operations can only be done when a particular quorum among replicas is gained. Such strategies can also be used to mask failures.

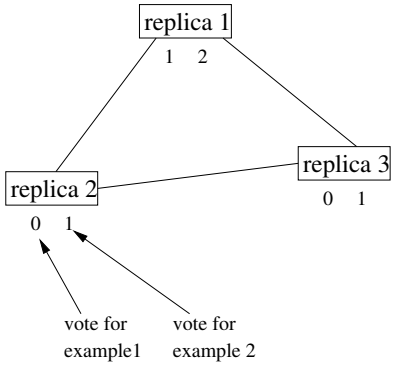
A vote can be regarded as a weighting to the desirability of using a particular copy. In other words, by assigning weights to a replica the popularity of a data item can be controlled and, hence, the update behaviour of the replicated system. In general, each read operation must first obtain a read quorum of R votes before it can execute the read operation on any copy. Similarly, each write operation must obtain a write quorum of W votes before executing the write. The original algorithm was developed for the maintenance of replicated files. The ideas can be extended to replicas as well and each file can be replaced by a replica. The algorithm is characterised by the following principles [47]:

- Every copy of a replicated file is assigned some number of votes.

- Every transaction collects a read quorum of R votes to read a file, and a write quorum of W votes to write a file, such that R+W is greater than the total number of votes assigned to the file.
- This ensures that there is a non-null intersection between every read quorum and every write quorum. Any read quorum that is gathered is guaranteed to have a current copy.
- Version numbers make it possible to determine which copies are current.

In other words, R and W have to be set for a group such that:
W > half the total votes
R + W > total number of votes for the group

An important feature of weighted voting is that groups of replicas can be configured to provide different performance or reliability characteristics. For instance, the reliability of write operations may be increased by decreasing W and similarly for reads by decreasing R. We use two simplified examples from [47] to explain the algorithm (see Figure 2.2).



	example 1	example 2
quorum size R	1	2
quarum size W	1	3

Figure 2.2: Example for a voting technique scenario

Let us first consider example 1. A quorum size for read means that a replica has to gain at least a sum of R in order to read a data item. Replica 2 has a vote of 0 and, hence, needs replica one in order to gain access to any data. The same is true for writing of data. This is exactly what the configuration is supposed to be: replica 1 represents a server and replicas 2 and 3 are weak clients. They can only access data if the server is available. On the other hand, a server can either read or write even if none of the clients is accessible. Example 2 is configured for a moderate read-to-write ratio. The local replica, replica 1, is assigned 2 votes and the remote replicas are assigned one vote. Although a read request can be served by the local replica only, for a write request one of the remote replicas has to be accessible.

A main disadvantage is the performance of a read operation since a quorum has to be gained. This is especially true if we compare it to the read performance of a lazy replication algorithm where a read can be done locally without any other replica participating.

The voting technique presented is a static one with a fixed quorum set and a static voting scheme. Recently, dynamic voting is used where a quorum set can be adjusted dynamically in reaction to system state changes [23] and [24]. The principle concept of dynamic voting is that it uses the current up-to-date copies at any time for deriving its dynamic quorum set. In [135] it is discussed why quorum systems are not very popular in commercial database systems. One of the shortcomings is the local read paradigm, i.e. even if you want to read data from a local replica, remote replicas have to participate in the read operation (a quorum is required). Another drawback is the read versus write ratio. If read operations occur more often than

write operations, the protocol should be optimised for read operations. However, it is also pointed out that the research community may be moving away from the local read paradigm. Additionally, differences of behaviours of protocols in LANs and WANs are mentioned.

2.5 State-of-the-Art in Replication

Based on the categorisation of replication approaches, we now select a few representative approaches that we discuss in more detail. We elaborate on related work in distributed database management systems and give an overview of the current state-of-the-art in data replication research focusing on asynchronous replication. The presented approaches deal with replication aspects in terms of update propagation, synchronisation and consistency. For read-only replication at the file level refer to Section 3.5. This is important to note since file level approaches in a Grid environment have different aims like speed in file transfer, security etc. A more detailed summary of state-of-the-art replication protocols in data management systems can be found in [116].

2.5.1 Discussion on Synchronous versus Asychnronous Replication

Synchronous replication mostly makes sense in small local-area networks with short latencies. In order to understand the problems that occurs with synchronous replication in more detail, we present the main ideas of [49]. This work also serves as a motivation that subsequent work discussed later in this section mostly focuses on asynchronous replication.

[49] defines eager and lazy replications in the following way: "When replicated, a simple single-node transaction may apply its updates remotely either as part of the same transaction (eager) or as separate transactions (lazy)." We see that we need two transactions for gaining a committed write on all replicas. If data is replicated to N nodes, a transaction does N times as much work because it has to be executed and committed at each node separately. Consequently, simple replication (transactional update-anywhere-anytime-anyway) behaves differently when the application is scaled (more lock contention) and global serialisability cannot be used in an efficient way.

[49] shows many analysis results and formulas for analysing eager and lazy replication and the authors mention that update-anywhere and lazy replication are more prone to serialisability violations than a primary copy approach. What is more, the analysis shows that deadlocks rise as the third power of the number of nodes in the network, and the fifth power of the transaction size. This is based on the assumption that eager replication contains no serialisation anomalies (inconsistencies) because locking detects potential anomalies and converts them to waits or deadlocks. However, eager replication reduces update performance and increases transaction response times since extra updates and messages are added to the transaction.

The authors also claim that eager replication is not an option for mobile and thus disconnected applications. The two major problems of synchronous replication are summarised as follows [49]:

- Mobile nodes cannot use an eager scheme when disconnected.
- The probability of deadlocks rises very quickly with transaction size and the number of nodes.

Furthermore, asynchronous update-anywhere (lazy group [49]) and an asynchronous version of the primary copy approach (lazy master [49]) are analysed. In the lazy group approach, when a transaction commits, a transaction is sent to every node to apply the updates. If two nodes update the same object, reconciliation is required. Time-stamps are one possible way to

detect the previous problem and reconcile the transactional updates. This can be done in the following way [49]: "Each object carries the times-tamp of its most recent update. Each replica update carries the new value and is tagged with the old object times-tamp. Each node detects incoming replica updates. The node tests if the local replica's times-tamp and the update's old times-tamp are equal. If so, the update is safe. The local replica's times-tamp advances to the new transaction's times-tamp and the object value is updated. If the current times-tamp of the local replica does not match the old times-tamp seen by the root transaction, then the update may be 'dangerous'. In such cases, the node rejects the incoming transaction and submits it for reconciliation." We can deduce that waits occur here more often than deadlocks. However, lazy group and lazy master still have to deal with deadlocks and much reconciliation overhead. Reconciliation is only used in lazy and not in eager replication schemes.

Besides analysing different replication modes, four goals are stated that an ideal replication scheme should achieve: availability and scalability, mobility (mobile computing: computers are not permanently connected to the network), serialisability and convergence. Convergence means that two data items that have different values initially must agree on the same value by exchanging messages. Normally, the most up to date value becomes the final value of both data items.

Conventional database management systems do not address the mobility goal very satisfactory. Consequently, [49] introduces a two-tier replication scheme to support mobile applications. In this scheme, two kinds of nodes exist: mobile nodes and base nodes that are connected permanently. In short, base nodes store replicas of the database and each object is mastered at some node. Furthermore, the scheme distinguishes between base and tentative transactions that work on local tentative data. This two-tier approach is proposed as a solution to the update synchronisation problem.

2.5.2 Lazy Update Propagation Approaches

We provide a short overview on lazy update synchronisation approaches. We do not go into detail but want to identify important work in the research community.

[2] proposes an *epidemic algorithm* for transaction processing. Using an optimistic precommit and conflict detection/resolution are part of the update algorithm. A general statement is "execute a transaction locally and commit it globally". Another epidemic approach can be found in [98]. In addition, epidemic models with fully replicated sites are discussed in [57] and combined with quora for updates and queries while maintaining global serialisability. The proposed quorum approach is supposed to be tolerant of site and communication failures. In [70] a replicated-object protocol for mobile and weakly consistent environments is presented. The protocol combines epidemic information propagation with voting. All replicas are equal (peers) and update everywhere is assumed in a replicated-object system.

[16] also proposes a lazy-master approach where all updates are directed to a primary copy. Global serialisability is achieved by a global concurrency-control mechanism that minimises co-ordination among sites. This approach is supposed to have fewer deadlocks than the one in [49], and the distributed transaction management problem is reduced to maintaining a global consistent graph called "replication graph". Furthermore, site failures are handled completely by relying on the local database system's recovery manager. A newer update replica approach based on a "copy graph" rather than a replication graph can be found in [17]. A performance study of the protocols introduced in [49] and [16] can be found in [7].

Incremental updates and reconciliation are discussed in [22]. The main problems that have to be solved are consistency regarding update propagation and transaction atomicity. Updates can be initiated at any single site and then propagated to the other sites immediately or delayed. If one site cannot accept the update, the other sites still continue with the transaction (non-

blocking behaviour). Since there is no atomic commit across all sites, this avoids sites getting blocked. Sites are brought incrementally to an agreement by using a reconciliation algorithm. A disadvantage of the proposal is the loss of strict serialisability but it is acceptable for many applications.

In [26] it is claimed that loose consistency can yield better responsiveness since waiting operations associated with multi-site commit protocols are avoided. The deferred update propagation model introduced in [26] is called a "strict primary copy approach" that can be compared to the primary copy replication in [16]. The use of the strict primary copy approach prevents update conflicts. This approach is supposed to be a very strict lower bound because it applies to even most restricted kind of transactions.

A lazy master replication approach with emphasis on efficient refreshment algorithms is given in [93]. In another project, Mariposa, a distributed data manager is presented, and it uses an economic model which is extended by replica management presented in [113]. An *anti-entropy* update propagation protocol for weakly consistent replicas is presented in [96]. It operates over diverse network topologies, including slow network links.

Approaches on Top of an Existing Database

All of the approaches presented above provide solutions inside the database management system and thus need to be included into the "database kernel". The following approaches build *on top of existing, commercial database management systems*, in particular, relational databases:

In [35] the Bengal database replication system is presented. Emphasis is put on remote, weakly connected mobile users (named *nomadic users*). It is claimed that client/server replication does not match the requirements of mainly disconnected users and thus Bengal uses a peer-to-peer relationship. An asynchronous replication method using "optimistic replication" is used where reconciliation of updates is done by special reconciliation servers. Like in conventional database approaches, version vectors are used to compare and exchange update information. Features of the proposed system also go back to the Ficus files system [94] that also provides an optimistic replication approach. Bengal was designed using Microsoft's Component Object Model (COM) and works for relational databases including Sybase SQL Anywhere, Oracle and Microsoft SQL Server.

Another approach in [128] first gives a taxonomy for replicating data stored in relational tables. The approach is based on and implementation on top of Oracle. In addition, an Adaptable Replication manager (ADR) is presented in [44] that builds on commercial distributed (relational) databases.

Related work in distributed database management systems and object replication can be found in [31]. A meta model for replication objects and data types in object-oriented databases is presented. The infrastructure sits on top of an existing object database management system. The application domain is business and economics, in particular for financial analysis in banks. The presented architecture tries to satisfy a variety of requirements. *Non-transparent replication* is used: replication users have to define explicitly which data items they want to access. Replicas are weakly consistent and update propagation can be delayed. Master-slave configuration is used and single objects are eligible for replication. Object references are addressed, too. Replication is realised by using a publish-subscribe mechanism. Update mechanisms use logging to store and later propagate changes to subscribed sites. Finally, an implementation of the system called Calanda is presented which is based on ObjectStore.

2.5.3 Group Communication Approaches

In the literature, several group communication approaches are presented which are optimised for LAN as well as WAN distributed databases. Typical group communication primitives

provide reliable message broadcast with delivery guarantees, atomicity and ordering properties. Here, we briefly state some representative approaches but do not address group communication further in this thesis.

[71] points out that atomic broadcast suffers from scalability problems as it involves coordination between sites before messages can be delivered. Research has been done in order to reduce the average delay for message delivery. A new atomic broadcast protocol has been produced which is based on optimistic assumptions about the network. The work focuses on clusters of work stations and more recent work can be found in [72].

Also [56], [115] and [3] deal with group communication. Whereas in the latter ones broadcast primitives are exploited and protocols are proposed, [56] gives performance measurements on a simulation of a replicated database using group communication protocols. A highly available Internet server is simulated to be replicated on a local-area network to serve Internet clients.

In [75] a broadcast protocol is proposed which assumes a hierarchical structure of nodes and is designed for wide-area networks and weak consistency replica management.

2.6 Commercial Database Management Systems

After the theoretical back ground in database theory and research, we now state briefly replication features implemented in some commercial database management systems. Since in our research phase we mainly worked with Objectivity/DB, we put most emphasis on this database management system and point out issues for object replication and then discuss briefly other object-oriented and relational database management systems.

2.6.1 Objectivity/DB

Since Objectivity/DB [89] is the main ODBMS system we are referring to in this thesis, we dedicate this section to explaining more details of this product and problems that we were confronted with when using Objectivity for wide-area data replication. Some of the problems mentioned are specific to all object-oriented data stores while others are Objectivity specific. However, the usage of an ODBMS allows us some simplifications like a global namespace and a global schema information.

Objectivity/DB is a distributed, object-oriented DBMS which has a Data Replication Option called DRO [90]. DRO is optimised for synchronous replication over a local-area network. In this section, we briefly describe DRO and its drawbacks, and state complications that one comes across when persistent objects have to be replicated.

Objectivity's Data Replication Option

DRO provides a synchronous replication model on the file level, i.e. entire Objectivity databases which are mapped to physical files can be replicated. Note that in the following section we use the term replica to refer to a physical instance (a copy) of an entire Objectivity file. There are basically two ways to use DRO: data can be written first, and then synchronised and replicated (**populate - replicate**). Multiple replicas can be created and synchronised and the data can be written into all the replicas at the same time (**replicate - populate**). Once a replica is synchronised with another one, all database transactions on a single replica are synchronised with other replicas.

Objectivity/DRO performs a dynamic quorum calculation when an application accesses a replica. When an application wants to access a replica (in Objectivity it is called image), a dynamic quorum calculation is initiated. Each replica is assigned a value. The quorum

calculation determines the amount of replicas that are available and adds their values. If the number of reachable replicas is greater than half of the total number of replicas, a quorum exists. Reading and updating a replica is only possible if a quorum exists, otherwise a transaction is not allowed to be executed. Since the absolute number of accessible replicas is not always a good way to assign values to a replica, it is possible to assign different weights to replicas in order to gain a quorum (majority of votes) in case of a network or site failure. By default, it is not possible to access data if a client does not have a quorum, but there exists a function that reads stale data even without a quorum. This bears the danger of reading out-of-date data. If a replica becomes inaccessible, the pages, which are written by accessible quora, are tracked and are automatically resynchronised as soon as the replica is accessible again.

There is one possibility to overcome the immediate synchronisation. A replica can be set to be off-line. However, only if a quorum still exists, data can be written to the other off-line replicas. There is no clean way in DRO to perform a real asynchronous or batch replication where you specify a synchronisation point in time. An asynchronous or batch replication method allows replicas to be out of sync for a certain amount of time. Reconciliation of updates is only done at certain synchronisation points, e.g. every hour. The lack of such an explicit asynchronous replication method is one of the reasons why DRO is not considered as a good option for WAN replication [102, 103] in a Data Grid (see Chapter 3) environment.

Like many commercial databases, Objectivity does not provide any optimisation for accessing replicas. The replica catalogue only contains the number of copies and the location of each single copy of a file. Once an object of a replicated file has to be accessed, Objectivity tries to get the first copy of the file which appears in the catalogue. It does not examine the bandwidth to the site nor takes into account any data server load. Some operations such as database creation require all replicas to be available. When we talk about Objectivity in this thesis we mostly ignore the fact that DRO exists and thus see Objectivity only as a distributed DBMS with single copies of a file. However, these single copies of a file can exist at distributed sites in the Data Grid and thus remote access to Objectivity data is possible.

Partial Replication and Associations

In Objectivity, replication is done on the file level and thus a file is regarded to be the smallest granularity of replication. Single objects are stored in containers. Several containers are stored together in an Objectivity database which corresponds to a file. Each object can have associations (also called links or pointers) to other objects which may reside in any database or container. Let us now assume that two objects in two different files are connected via one association. When an Objectivity database (a file) is replicated, this association gets lost when only one of the files is replicated. Note that there is still the possibility of remotely accessing objects. Hence, the replication decision has to be made carefully and possible associations between objects have to be considered. This may result in replicating a set of files in order to keep all the associations between files. Furthermore, this also imposes severe restrictions on partial replication where only particular objects of a file are replicated.

Definition: When a certain set of objects is selected which does not have associations to objects which are not in the set, the replication set is *association safe*. This is a particular restriction of object-oriented data stores and does not only hold for Objectivity.

The association problem is also an important difference and complication compared to a relational DBMS. Whereas in an ODBMS an object can be stored in any file, in a relational database data items are stored in tables that have well defined references to other tables.

The Objectivity File Catalogue

The integration of files into the native Objectivity file catalogue is done with a tool called *ooattachdb* which adds a logical filename and the physical disk location (physical filename) into the catalogue. Furthermore, it guarantees that links to existing Objectivity databases are correct. The schema of the file is not checked. This feature is used when a file created at one site has to be integrated into the file catalogue of another site.

Since the native catalogue only has a one-to-one mapping from one logical to one physical file, replicas are not visible to the local site (not taking into account DRO). Furthermore, it is possible to have single links from one site to another. For instance, site 1 has a file called X and this file shall be shared (not replicated) between several sites. The filename and the location can be integrated into the local file catalogue and a remote access to the file can be established. Note that this requires an Objectivity Advanced Multi-threaded Server (AMS) [53] running or the usage of a shared file system like AFS or NFS that connects both sites. The AMS is responsible for transferring (streaming) objects from one machine to another and thus establishes a remote data access functionality. We address the more general solution where no shared file system is available.

In the High Energy Physics (HEP) community (see Chapter 3) it is generally agreed and proven that DRO is not optimised for the use in a WAN. Hence, all our discussions here neglect the DRO of Objectivity/DB and we conclude that the current implementation of DRO is not a feasible solution for the HEP community.

2.6.2 ObjectStore

ObjectStore is an object database and uses an asynchronous approach which is similar to the primary copy approach. One replica is referred to as the master and all the other replicas are slaves. Updates can only be executed by the master replica whereas all the slaves are read only. The replication process is triggered by a batch job that is initiated at a certain time that can be changed by the database user. For instance, updates can be propagated each hour (i.e. update propagation time is 60 minutes). However, the time value can be set to 0 as well which corresponds to a synchronous replication approach.

2.6.3 Versant

Versant is another object database management system that provides both synchronous and asynchronous replication of objects although asynchronous replication is recommended like in all the following database management systems. Unlike Objectivity, Versant's asynchronous replication is supposed to be optimised for local-area networks (default replication mode) as well as wide-area networks (advanced replication mode). Furthermore, master-slave as well as peer-to-peer configurations are possible. In Versant Asynchronous Replication 6.0 transactional (objects of the same transaction are replicated) as well as batch replication (objects of several transactions) are supported.

Replication can either be automatically or explicitly triggered. In the first method, event notification is used. In the second case, C++ or Java API are used. Internally, Java serialisation coupled with GZIP compressions is used for exchanging replica update information over Java TCP/IP sockets.

Since asynchronous replication sometimes creates conflicts if the same objects are updated by several users at different sites at the same time, conflict detection and resolution methods are provided.

2.6.4 Oracle

As opposed to the three object databases above, we now discuss relational database management systems and their replication features. We refer to Oracle *9i* and the pure relational database rather than the object-relational interface.

Oracle supports synchronous and asynchronous replication features. Whereas Objectivity provides a quorum system for updates, in Oracle's synchronous replication model updates can only succeed if all replica sites are available and take part in the two-phase commit protocol for updates. As regards asynchronous replication, Oracle also supports master-slave replication. Furthermore, conventional tables as well as materialised views can be replicated where a materialised view is typically refreshed periodically. Asynchronous replication is a transactional replication mechanism with deferred transactions (expressed in remote procedure calls), i.e. they are stored locally and then forwarded to other replica sites. Thus, it is also called "store-and-forward" replication.

Similar like all other products offering asynchronous replication tools, Oracle also provides tools for conflict detection and resolution.

2.6.5 Sybase

Similar to Oracle, also Sybase's Replication Server 12.0 provides synchronous and asynchronous replication where the latter is based on transaction logs. A publish-subscriber model allows that only sites that are interested in published information receive changes of replicas.

As regards asynchronous replication, a master-slave replication model guarantees that updates are done asynchronously only on master replicas and thus the owner of the information can control access to it.

2.6.6 DB2

IBM's DB2 replication tool DataPropagator again has similar features like the products above and provides asynchronous replication with a log-based change-capture component.

2.6.7 Mircosoft SQL Server

Like Sybase, also SQL Server provides a subscription model to replicated data. In more detail, the following types of replication are provided:

- Snapshot replication: This is the simplest form of replication that takes a picture or snapshot of a database at a given moment. Replication can either be scheduled or done on demand (like in most other systems, too).
- Transactional replication: This is the conventional synchronous replication model of transactional data.
- Merge replication: This form of replication allows for the highest level of autonomy and thus satisfies disconnected users. Changes are merged with replica sites when a disconnected user connects again to the replication environment.

Chapter 3

Data Grids and Data Management Challenges

This chapter provides a problem specification of Data Grids with emphasis on data management and replication. In particular, motivation for Grids and their use in High Energy Physics are discussed.

3.1 The Vision of a Grid

Grid computing can be regarded as a special form of distributed computing where computing and storage resources are distributed over several locations (sites) that are connected via wide-area network links. A site normally has a local-area network which itself has distributed computing and data storage resources.

Grid computing in general comes from high-performance computing, super computing and later cluster computing where several processors or work stations are connected via high-speed interconnects. The term "Grid" has first been introduced in [39] and has gained popularity in the above mentioned communities. Currently, there is an entire Grid community, several Grid projects are established (see later sections for details) and a common Global Grid Forum [48] is in place.

"The grid is an emerging infrastructure that will fundamentally change the way we think about - and use - computing" , these words are used in the preface of [39] and an analogy to the electrical power grid has been made in order to illustrate the potential of the Grid. Its main idea is to bring computing resources virtually together and use them as a single, large computer. Another terminology for this is *Computational Grid*. Currently, computational Grid applications are mainly found in scientific environments and not so much in industrial ones. Recently, the largest Computational Grid application is SETI@home [112] that uses several thousand CPUs all around the world where all machines contribute to a large computational task.

Grids do not only require computing resources connected via local and wide-area network connections but rely on specific software that makes use of these distributed resources. In general, Grid middleware is used to bridge the gap between the end user application and the distributed computing resources. Several Grid projects are currently available that produce and/or offer middleware software like Globus [41], Legion [51] , Condor [10] etc. Computational Grid software mainly addresses job scheduling and resource allocation as well as some reservation of computing resources. Network bandwidth reservation and quality of service are still under development and not really functional yet.

Originally, Grids were intended to deal with large computation intensive science and en-

gineering applications also referred to as "Grand Challenges" which cover examples as follows [39]:

- batch physics simulation and multidisciplinary simulations
- interactive applications in several domains
- rendering and signal processing; complex analysis and data from large, distributed archives, etc.

In the last item we see already that large computational tasks might require large input data sets that are distributed to several sites. Once data amounts are very large and in Tera- or even Petabyte scales, special data management tasks like placement and replication of large data sets need to be solved. This kind of Grid where data is distributed over several sites all over the world is called *Data Grid*. In a Data Grid, there are still large computational tasks to be solved but data management is a major factor for serving applications.

3.2 Data Management Problems

Data Management problems appear in several kinds of Grids (Computational and Data) but are tackled in different ways depending on the data size and the purpose of the Grid. In [118] we provide a detailed discussion and outline some major points here.

Basically, we can distinguish two kinds of Grid applications: *high-performance* and *high-throughput* applications. A typical high-performance application tries to maximise the resource usage for a single application in a way that this single application is executed as quickly as possible. High-performance thus is an indicator for how long a single application runs in order to complete its tasks. Mainly, high-performance applications can be split into several subtasks which are executed in parallel. In contrast, in high-throughput computing, several entire applications are executed and a "fair" schedule gives preference to several applications rather than to a single one. Thus, the overall throughput of multiple applications is optimised.

Data management requirements are different for high-throughput and high-performance computing tasks depending if data is persistently stored on secondary (disks) or tertiary (tapes) storage devices. As identified in [118], several common operations for data processing and management include: data pre-processing, data fusion, data storage, data analysis, query estimation and optimisation as well as visualisation, navigation and steering. In high-performance scientific computing, data access is generally rather regular, such as within a loop of a numerical calculation, for instance. However, data access patterns are often very difficult to determine as regards the high-throughput applications in High Energy Physics. The unit of data transfer and the amount of data to be transferred have a significant influence too. Furthermore, computing intensive high-performance applications tend to store rather small amounts of data compared to their computational effort.

In [118] we identify several data management core services that include data access operations, location transparency and global name space management, security management as well as persistence and replication. Our main interest in data management is the replication aspect where large amounts of data are replicated within several sites distributed in a world-wide Data Grid. We now focus on high-throughput computing in a Data Grid.

3.3 Data Management Challenges in High Energy Physics

The computing models of some next-generation High Energy Physics experiments include several data management challenges and thus is sometimes referred to as a distinct example for

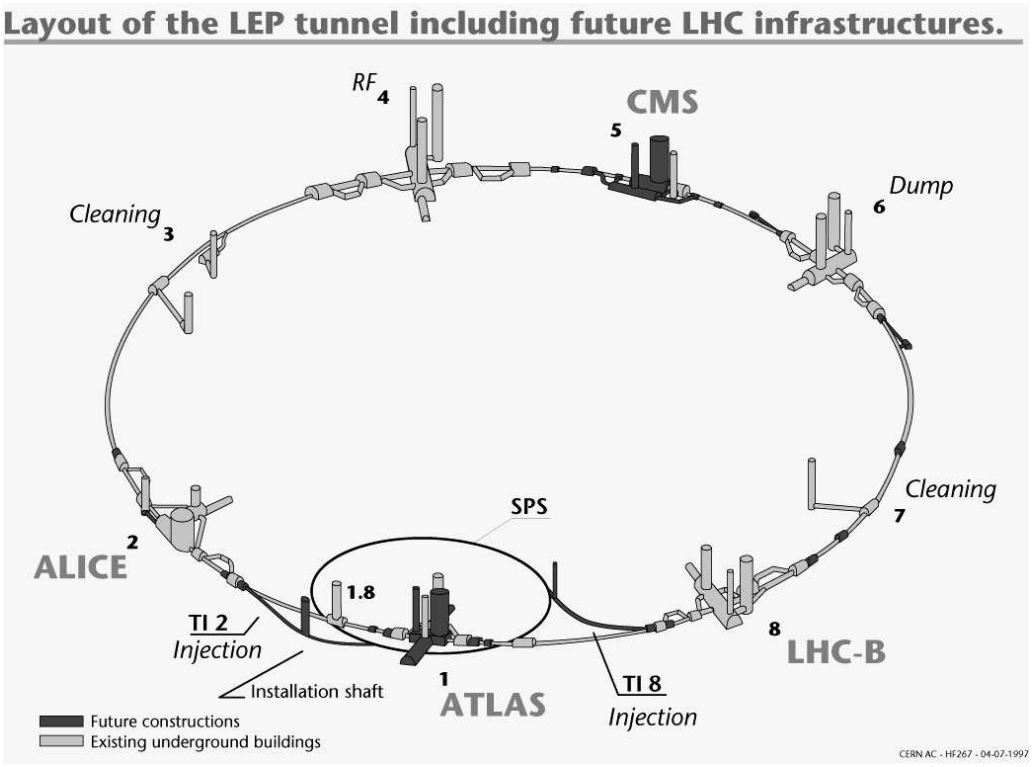


Figure 3.1: The Large Hadron Collider and the four LHC experiments: ALICE, ATLAS, CMS and LHCb

a world-wide distributed Data Grid. We now give some background in order to justify this assumption.

3.3.1 CERN and High Energy Physics

CERN, the European Organization for Nuclear Research, is well known in the computer science community for the invention of the World-Wide Web which goes back to Tim-Berners Lee. However, the main task of CERN is not computer science research but physics research to discover the origin of matter. Some details on how CERN is achieving the physics and computational challenges are given in this section.

CERN is an international organisation with 20 European member states and several non-member states on the American, Asian and Australian continent. Around 6,500 researchers from all over the world representing over 500 universities and about 80 countries participate in the research effort to discover new matter and the origin of the universe.

CERN hosts several High Energy Physics (HEP) experiments that use large particle accelerators and detectors for physics experiments. Until the end of the year 2000, the world's largest particle accelerator LEP (Large Electron Positron accelerator) was in place with a circumference of 26.7 km. Starting from 2006, a new accelerator LHC (Large Hadron Collider) will be in operation in the same tunnel about 100 meters under the earth where LEP was until recently. With LHC, four experiments (physics collaborations) will start using next generation particle detectors (see Figure 3.1). Each of the four collaborations (ALICE, ATLAS, CMS and LHCb) will produce a detector that will be placed at four different locations of the LHC.

Whereas CERN has a cyclic accelerator, High Energy Physics experiments can also use linear accelerators and then use a single detector. SLAC [114], the Stanford Linear Accelerator Center which is operated by the Stanford University in California, is one the most famous

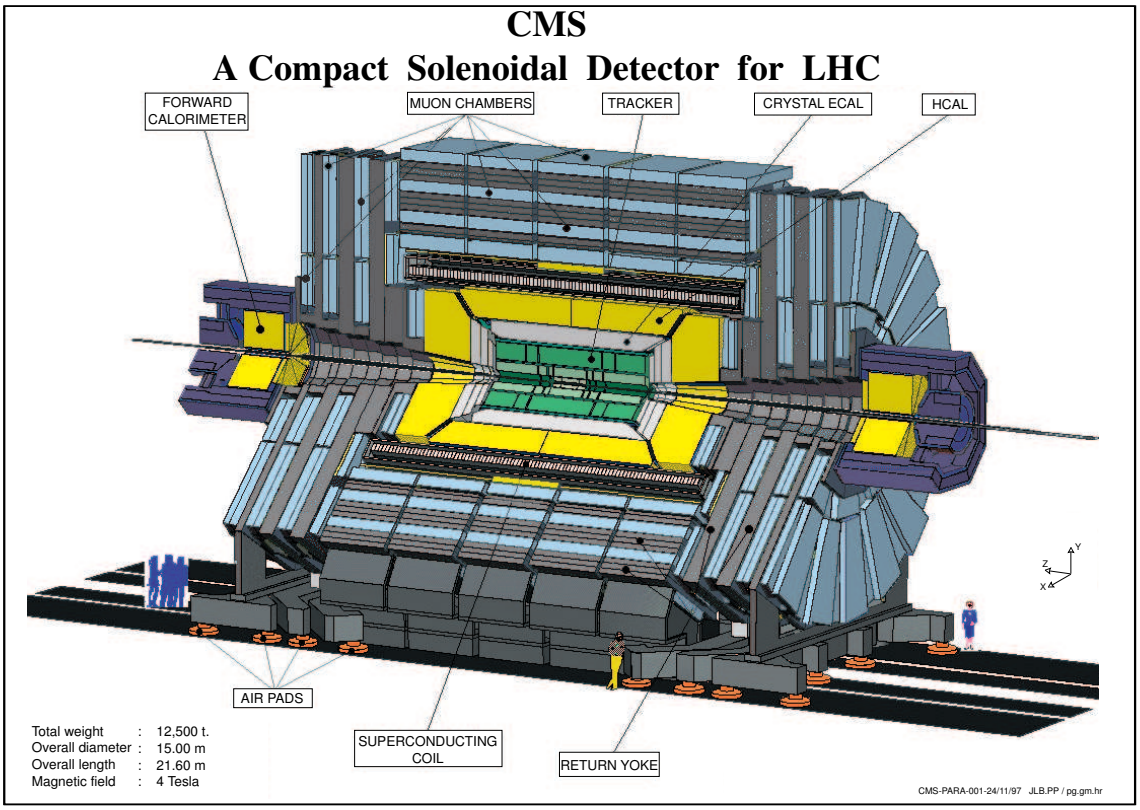


Figure 3.2: The CMS detector

representatives in this field.

Each of CERN's four LHC experiments will start to produce several Petabytes of data a year over a life time of 15 to 20 years. Since this amount of data was never produced before in the HEP community, special efforts concerning data management and data storage are required. Details on the CMS experiment are given in Section 3.3.2.

All these four experiments have in common that hadrons, sub-atomic particles, are accelerated in the Large Hadron Collider to almost speed of light in order to collide in the detectors. In detail, the detector measures and records what happens during a particle head-on collision: particle tracks, the creation of new particles, their energy values etc. are recorded at very high data rates.

3.3.2 The CMS Experiment and the Data Model

The work for this thesis was carried out in the CMS (Compact Muon Solenoid) experiment and thus CMS and its data model are described in more detail. The other three experiments have rather similar data models and data challenges and thus CMS is representative of the LHC experiments.

Recording and storing data is a rather complicated chain and we now give a simple view from a computer science but not physics point of view. A particle collision takes place in the middle of the CMS detector (see Figure 3.2). The detector itself consists of several sub-detectors that use different detection technologies.

The collision of particles in the detector is called an *event* in physics terminology. During the event, several new particles are created and go through several layers in the CMS detector. Collisions will take place at the rate of 40 MHz, i.e. particles accelerated in the LHC will collide

40,000,000 times a second in the detector and 40,000,000 events are detected. Since the result of a collision can never be predicted with 100 per cent accuracy, recorded data can either be interesting for a physicist (a signal is found) or not interesting (only background is produced). Consequently, a filter system is required to filter signal from background. In addition, since the amount of data to be recorded and stored in electronic format would be too big, the filter system also reduces the amount of data to be finally stored on disks or mass storage systems.

CMS uses two main computing systems in the data processing part for event filtering and final data storage: the *online* and the *offline system*:

- online: The main task of the online system is to reduce the number of events from 40,000,000 per second to about 100. This is called the output rate of the online system which in turn is the input rate for the offline system. The data reduction task will be done using a three-level trigger system [27] where the first trigger level is completely implemented in hardware. The other two trigger levels are hardware and software based.

  The online system produces output for 100 events per second which then have to be stored permanently on disk or tape by the offline system. A single event is called a *raw event*.

- offline: The offline system has basically two major tasks. The first is to store raw events produced by the online system at a rate of 100 Hz. A single raw event will have a size of about 1 MB and consequently the offline systems needs to write data with a speed of 100 MB/s. An event is the unit for storing data. The second major task of the offline system is to do some calculation on each raw event and produce the so called *reconstructed event*.

In the remainder of this thesis we only consider offline tasks, in particular data distribution and replication and do not take into account event filtering nor local storage technologies. For a detailed discussions on local CMS storage management refer to [61].

We now take a closer look at the CMS data model and the amount of data to be stored. Basically, a hierarchical data model can be identified where the amount of data is decreased with an increase of information in the data. The hierarchy basically contains the following three types of data:

- *raw data*: Raw data is the actual data stored after the online filtering. A single raw event will be stored on disk and will have a size of about 1 MB.

- *reconstructed data*: Raw data events do not contain much information that physicists can use for data analysis in their search for the origin of matter. Thus, specific reconstruction functions and algorithms are applied on raw data events and detailed information about the event is stored in reconstructed events. The size of a single reconstructed event is smaller than that of a raw event and is between 100KB and 500KB per reconstructed event. Reconstructed data contains information about particle tracks in the detector, energy values etc. and consequently structure is brought into the raw data.

- *tag data*: Tag data is summary data that summarises information stored in raw and reconstructed data. The size of a single tag data event is about 100 bytes.

For completeness and mutual understanding of terms used in the physics community and in other LHC experiments, we introduce the terms *Event Summary Data (ESD)* and *Analysis Object Data (AOD)* as mentioned in [80]. In principle, reconstructed data can be divided further into ESD and AOD where ESD has a size of about 200 KB and an AOD event about 10 KB. AOD is produced by additional reconstruction steps from reconstructed data. This further distinction between ESD and AOD is useful for end user physics analysis.

CMS is supposed to take data 100 days a year with the offline data rate of 100 MB/s. In total, the amount of raw and reconstructed data produced a year is in the order of a few Petabytes. Neither the correct amount of data to be produced per year starting from 2006 nor the correct data size for the different data types can be given now but a rough estimation which reflects the current understanding of the data sizes is given in Table 3.1.

data type	event size	storage amount per year
raw data	1 MB	1 PB
reconstructed data	200 KB	200 TB
analysis object data	10 KB	10 TB
tag data	100 B	100 GB

Table 3.1: Different data types and estimated storage amounts. Note that the numbers here are average event sizes.

The reconstruction process in the offline system requires lots of computing power (for details refer to [132]) and clusters or farms of workstations will be used to achieve the computing requirements. Although the CMS detector produces data in the vicinity of CERN, data is transferred via high-speed networks from the detector to the computer centre and a large amount of offline computing resources will be available in the CERN computer centre. However, the computing power at CERN will not be sufficient and external computing centres will be included into the computing infrastructure. Section 3.3.3 outlines the challenges that CMS is faced with as regards data distribution to world-wide distributed computing centres.

CMS uses an object-orient software development processes for writing physics code. In particular, reconstruction algorithms and the physics software framework are written in C++. Major parts of the software have already been done but the software development effort will continue for another few years. Events (raw, reconstructed and tag data) will be stored in databases but the finial decision for a certain database technology has not been made yet and will be due to the end of 2002. Currently, an object-oriented database management system is used to store simulated data. Storing data in objects is an agreed solution since it fits naturally to the object-oriented software development process. Consequently, objects are the basic building blocks for physics data. Note that database technologies are not used by all four LHC experiments which partly rely on object streaming technologies[1]. To sum up, for the CMS experiment the final decision about the DBMS (object-oriented or relational, which vendor, etc.) has not been made yet, but the current software development processes uses an object-oriented approach and Objectivity for storing data persistently.

An important feature of data in High Energy Physiscs is that it is mainly **read-only**. This is true for most of the event data types discussed so far. In contrast, additional data sources are *calibration* and *conditions* data that contain meta data for event data and are subject to constant changes. In general, new data is either added or new versions are created rather than updating existing data.

Once data is stored in a database, a physicist can start his *physics analysis*. This can be regarded as one of the most important tasks of the physicists after the data has been detected and stored. Basically, a physics analysis job consists of several steps. In the first step, an initial collection of events is chosen that fulfils certain physics properties. In the next step, several selection criteria (queries) are applied in order to reduce the data set. We report on a more detailed physics analysis example in [111]. More comprehensive background on physics

[1]For some specific data types which are used mainly for detector construction, it might be useful to extract small amounts of one data store, serialise them and provide them in a different data format. We have done preliminary work on serialising objects stored in Objectivity using XML to serialise them [13].

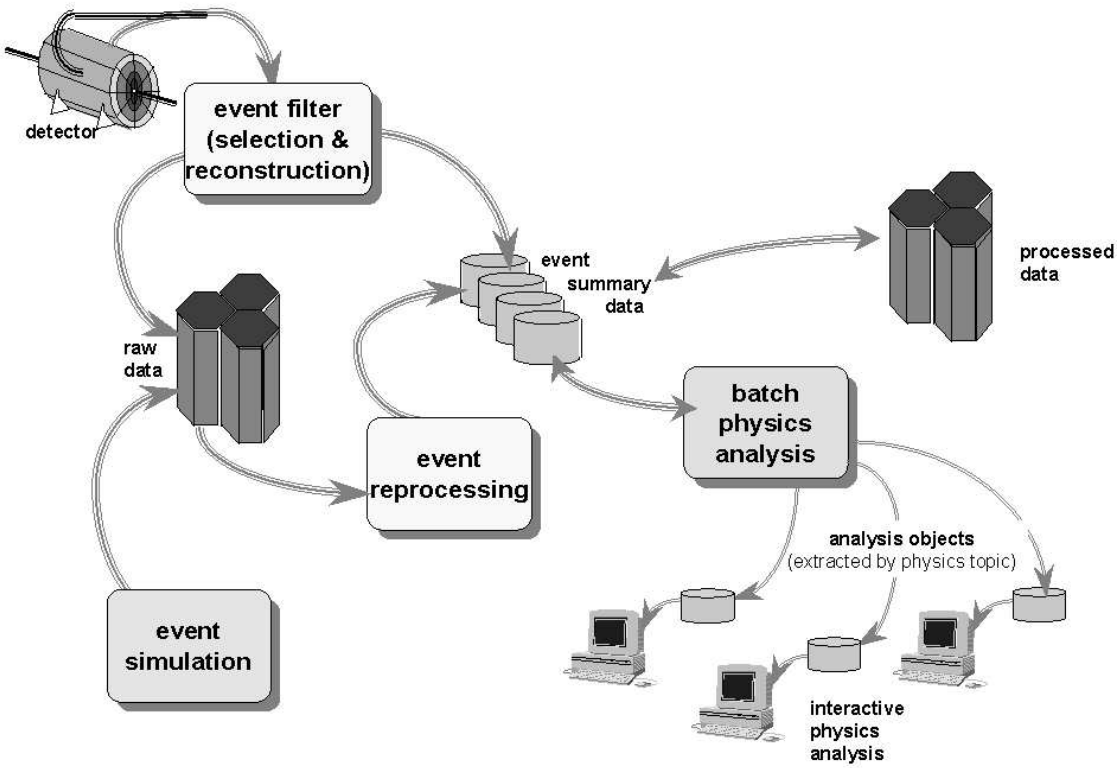


Figure 3.3: Simulation and data reconstruction chain

analysis can be found in [61].

Since currently the CMS detector is not yet in operation, event data is simulated and afterwards generated. This simulated data is then reconstructed like "real data" and used for physics analysis. Figure 3.3 shows the entire data acquisition and reconstruction chain including simulation of data. Currently, production runs are done with simulated data in the order of several Terabytes.

3.3.3 Data Challenges: Distribution and Replication

The amount of data to be produced is rather similar in all the four LHC experiments and is in the order of a few Petabytes a year. Since computing resources at the CERN computer centre will be limited, LHC experiments will work with computing models where computing and storage resources are distributed to several computing centres of CERN member and CERN non-member states in the physics collaboration.

Since LHC experiments like CMS are collaborations of over a thousand physicists from many different universities and institutes all around the world, the experiment's data will not only be stored locally at CERN but parts of the data will be stored at world-wide distributed sites in so-called *Regional Centres* and also in some institutes and universities. The distributed computing model of a typical LHC experiment is shown in Figure 3.4.

These Regional Centres are part of the distributed computing model and should complement the functionality of the CERN centre. The aim is to use computing power and data storage in these Regional Centres and allow physicists to do their analysis work outside of CERN with a reasonable response time to their data rather than accessing all the data at CERN. This should also help the collaboration to have many scientists working spread around the world. Regional Centres will be set up for instance in Italy, France, Great Britain, USA,

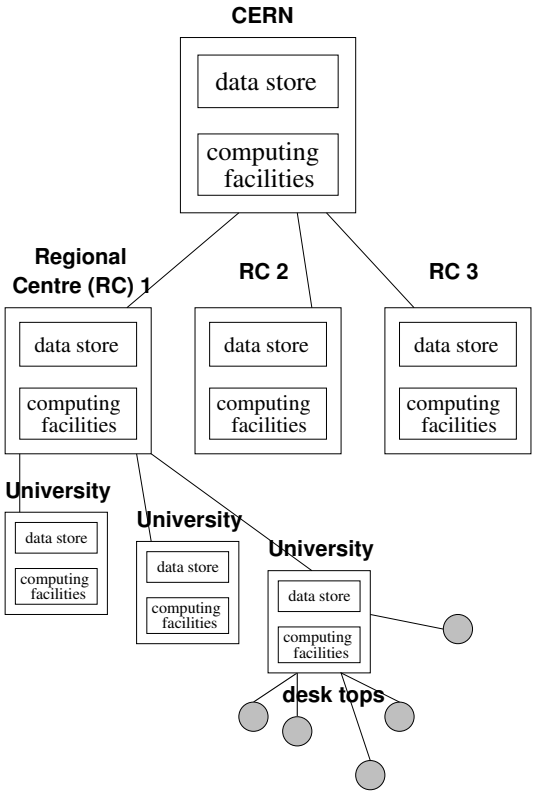


Figure 3.4: Example of the network of one experiment's computing model

Japan etc.

In the CMS experiment, the distributed computing model has been discussed in an early technical proposal [27] and more advanced models for the four LHC experiments have been studied in the Monarc project [80] at CERN.

Since Regional Centres will provide data and computing resources, large data volumes need to be replicated to Regional Centres where reconstruction and analysis can be done. Regional Centres are arranged in a hierarchical way with several tiers where CERN is a tier 0 centre storing all raw and reconstructed data. In a few countries (in the order of 5) tier 1 Regional Centres will be located each storing about a third of CERN's data volumes. Thus, roughly 300 to 500 Terabytes of data need to be transferred (replicated) between CERN and a single tier 1 centre. A tier 1 centre is supposed to be a large computing centre serving one or more countries. Several tier 2 centres will be established within a single country and are normally situated in universities and institutes. One may further distinguish a third and a forth tier in the hierarchy. The main idea is that the last tier (either 3 or 4) are individual desktop machines of physicists doing physics analysis.

This hierarchical structure, the data distribution and replication as well as the distributed computing environment contribute to the fact that the High Energy Physics community can be regarded as a distinguished example for a Data Grid. We define a typical Data Grid as a set of sites (computer centres or small local-area networks) connected via wide-area networks. Large, data and CPU intensive high-throughput applications are running on the distributed sites and networks connections have high data traffic.

Based on the large amounts of data to be transferred in the Data Grid, i.e. between the distributed tiers, networks can be a bottleneck for data transfers. Let us consider a 622 Mbps ATM network link between CERN and a Regional Centre. This corresponds to a maximum

transfer rate of 77.75 MB/s which cannot be reached since this is the theoretical limit of the network link. If we further assume a rather conservative average bandwidth of about 30 MB/s [119], roughly 2.6 Terabytes of data can be transferred per day over a single network link. However, since the capacity increases and the latency of network links decreases every year, wide-area networks will most probably not impose inherent bottlenecks anymore for data transfer. Current estimates foresee OC-48 network links between CERN and tier 1 sites. A detailed cost model for data storage and distribution is proposed in Section 4.2.

3.4 The European DataGrid Project at CERN

Since all the for LHC experiments at CERN have similar computing models, problems and challenges, mutual computing projects have been established. A project called RD45 [101] provided research and development for world-wide distributed databases and data management. The original focus was on object storage technology and all four LHC as well as other HEP experiments from different physics laboratories like SLAC (California), Fermilab (Chicago), DESY (Germany) were involved. Whereas RD45 concentrated on storage technology, the Monarc project [80] focused on models for network utilisation and the hierarchical structure of Regional Centres. Monarc can be regarded as a predecessor of a Grid project.

3.4.1 Aim and Background

Based on the need for mutual projects, common computing solutions and sharing of computing expertise, the European DataGrid Project [33, 64] (note the spelling "DataGrid") has been discussed, established and initiated at CERN in collaboration with external participants in early 2000 and officially been started on 1 January 2001. The emerging Grid technology and the vision of a Grid are promising for LHC experiments. In this chapter we only give an introduction to the scope of the project but architectural proposals and solutions are given in Chapter 4.

The European DataGrid project is funded by the European Union for a period of three years, starting from 1 January 2001. The project contains three application domains:

- High Energy Physics applications: these are the four LHC experiments ALICE, ATLAS, CMS and LHCb
- Earth Observations: this covers the European Space Agency (ESA)
- Biology: basically bio-informatics applications

The main aim of the project is to solve large, data intensive computational tasks as partly outlined in previous sections. Thus, job scheduling, data distribution and replication, information and storage management are of main interest. In order to solve these core tasks, the project is organised in several work packages as illustrated in the next section. Since the tasks have dependencies among each other and need close interaction, the work is split into several work packages to focus on certain tasks. This thesis focuses on replication issues in High Energy Physics applications but general results are also applicable in the other two application domains.

3.4.2 Work Package Categorisation

A work package (WP) can be regarded as a sizable sub-task in the entire project needing a substantial team of people. The categorisation starts with *five middleware work packages* that actually design and develop software and services:

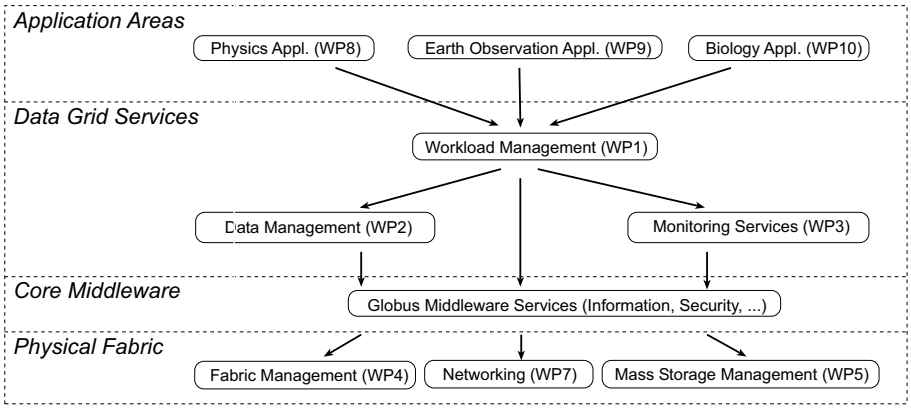


Figure 3.5: Overall interaction of the project work packages

- *Workload Management (WP1)* defines and implements components for distributed job scheduling and resource management.
- *Data Management (WP2)* deals with distribution, replication and optimised access to data in the Petabyte range. Architectural solutions are given in Chapter 4 and are partly within the scope of this thesis.
- *Monitoring Services (WP3)* cover application monitoring and Grid wide information services.
- *Fabric Management (WP4)* provides tools for configuring and managing computing centres hosting clusters of several hundred or even thousand nodes.
- *Mass Storage Management (WP5)* deals with interfaces to mass storage systems consisting of disk and tape storage.

Middleware software produced by these five work packages is integrated and tested by the *Testbed (WP6)* and can then be used by the three applications (also organised in work packages: *HEP Applications (WP8)*, *Earth Observation Applications (WP9)* and *Biology Applications (WP10)*) in a quasi production environment. Another work package, *Network Services (WP7)*, deals with wide-area networking issues. Finally, project dissemination and project management are done by *Dissemination (WP11)* and *Project Management (WP12)*, respectively.

Globus [41] has been chosen as the Grid middleware software and several development efforts of the work packages focus on some Globus services. The DataGrid project has good working relationships with the Globus team and some architectural decisions are done in common. A sketch of the work packages and the possible interaction is given in Figure 3.5.

3.4.3 Related Data Grids in the High Energy Physics community

Data Grid concepts are being explored in a number of projects world-wide. In the U.S., the Earth System Grid is applying Data Grid technologies to the management of climate data [5] while Particle Physics Data Grid (PPDG) [95] and Grid Physics Network (GriPhyN) [50] are both working in the HEP domain.

PPDG and GriPhyN are the most related projects in the HEP domain (GriPhyN is also addressing requirements of sky survey and astrophysics applications). In principle, it is desirable

that the three Data Grid projects in HEP, DataGrid, GriPhyN and PPDG, work closely together and have inter-operable architecture and software. Some of these aims have already been achieved in the GDMP [108] (see also Chapter 5) project since some mutual code development has been done.

GriPhyN is mainly addressing fundamental IT research focused on realising the concept of *Virtual Data*. Virtual Data in general means that data does not necessarily have to be available in a persistent form but is created on demand and then materialised when it is requested. In this virtual data space, requests can be satisfied via direct access and/or computation, with local and global resource management, policy, and security constraints determining the strategy used.

PPDG is a project that deals only with High Energy Physics applications but focuses on more immediate issues relating to file replication, job scheduling, and so forth.

For a more complete summary of current Data Grid project refer to [91].

3.5 File and Object Replication

In Chapter 2 we discussed replication in database environments and thus dealt with replication and update synchronisation of single data items. However, replication is also used in other distributed systems like distributed object systems, distributed file systems and for replication of Internet contents. Whereas distributed object and database systems deal with replication of data items or objects, in a file system the lowest granularity of replication is a file. In this section we briefly discuss the impact of the level of replication and outline general replication approaches in distributed systems including Internet replication.

The granularity of replication has an important impact on the replication strategy, the complexity and the protocol used. The granularity also depends on the distributed system that uses replication.

Definition : The *granularity of replication* defines at what level a replication process takes place. A file level granularity replicates entire files without any knowledge about the schema or structure of data items (objects) in the file. On the other hand, object granularity regards several bytes within a file as objects and can replicate single objects of an entire file.

An even higher level of granularity is to treat several files as a collection or set and replicate entire collections. In general, if the update frequency is low or even 0 (read-only data), a replication protocol does not have to take into account update synchronisation problems and thus the complexity of the protocol is low. The complexity is increased if the update frequency and the replication granularity increase. Figure 3.6 depicts the relation between replication granularity, protocol complexity and update frequency.

In databases replication, as discussed in Section 2.5, the replication granularity is on the data item level. In the Grid environment, often file level granularity seems to be sufficient in many cases but there is still a need for lower granularities in particular cases (see Section 3.5.1). In this thesis, we mainly focus on file level replication and later discuss update synchronisation of database files. When we speak about object-level replication, we assume that the data model and the schema are known to the replication mechanism. This is important for update operations.

3.5.1 Motivation for Different Replication Granularities in HEP

In the DataGrid project we mainly deal with file replication since the application software (provided by the physics experiments) will provide the necessary mapping of single objects to

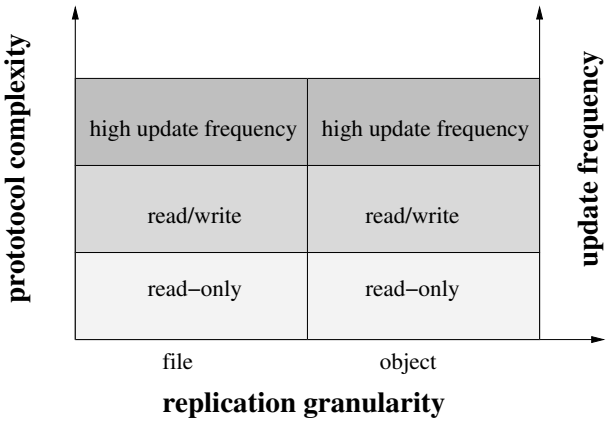


Figure 3.6: Granularity of replication compared to the replica update frequency and the protocol complexity

files. Thus, most of the Grid middleware software does not need to know about the internal data structure in the files.

However, at the highest level of abstraction in the experiments' data models, data consists of persistent objects and can be accessed through an object-oriented navigation mechanism. Files appear only at lower abstraction layers as implementation mechanisms for this object view. In [120] we discuss in more details that object level replication is desirable for some applications in High Energy Physics, in particular for end-user physics analysis.

An object-level replication approach is considered in [120]. In this approach, the Grid services also provide the navigation mechanism by which the applications can access the objects, so allocation of objects to files can be fluid and under Grid control. In more detail, if the number of requested objects of a file is small, requested objects are copied into a new file and then replicated to the destination site. For more a detailed discussion and a prototype implementation refer to [120] and [62].

3.5.2 Distributed Object Systems

Distributed object systems in general provide access to transient or persistent objects distributed at remote sites. CORBA [87] is one possible standard for a distributed object system. Object replication is available for fault tolerance but is not part of the CORBA standard. CORBA in some way addresses similar problems like a distributed database management system: although normally CORBA objects are transient and not stored permanently, a transaction system exists to handle read and write requests to distributed objects. Replica management in CORBA based on interceptors is discussed in [81].

In contrast to Globus, Legions (a Grid middleware and sometimes also called distributed operating system) deals with distributed objects rather than files. However, in Legion the replication of objects is possible. It is supposed to be rather easy to add but not implemented yet [82].

3.5.3 Distributed File Systems

In the context of this thesis distributed file systems are of more interest than distributed objects systems and thus need a more detailed discussion. In principle, file replication in a Data Grid has some commonality with replication in a file system. As compared to a database

management system, a file system does not have such a strong control over access to data, i.e. read and write operations are not under strict control of a data server.

One of the main distributed file systems for wide-area networks is AFS [66] but it does not provide file replication features which is the case for Coda [110, 14]. A newer project that has its origin in the Coda project is the InterMezzo File System [15] which is a client server file system that maintains replicas of file system data across multiple server machines. The file system can also be used for replicating HTTP servers via fail-over replication. Intermezzo does not entirely replace a disk file system but wraps around an existing one.

Ficus [94] is another distributed file system for wide-area networks supporting replication. Similar to lazy database replication, Ficus provides an optimistic update replication process where updates are synchronised at pre-defined points in time. It uses a *single-copy availability* mechanism that allows file updates even if only a single replica is available and no quorum is needed. This is in contrast to a one-copy serialisable update mechanism. Unlike Intermezzo and Coda, Ficus is a peer-to-peer file system where all clients and servers are equal.

3.5.4 Replication and Caching in the Internet

Since Data Grids mostly rely on Internet protocols and services, we briefly discuss Internet based replication and caching techniques and regard this as a related domain to Data Grids. Replication granularity is normally a file (text, graphic, etc.).

Replication is sometimes mentioned to be a form of caching: data is kept redundantly and made available on demand when requested more often. Whereas caching normally keeps redundant data for a short amount of time, replication keeps data longer.

In the Internet, often the same data is requested and thus transmitted repetitively. Web caching tries to keep data locally in order to reduce "wasteful use" of wide-area networks [11]. Mirror sites are commonly deployed to remedy this problem. However, mirror sites are static and do not support redirection of requests: the user has to determine which mirror site to use. Additionally, consistency and correctness of replicas at mirror sites is generally not known [11]. The Internet2 Distributed Storage Interface (I2-DSI) project proposes an architecture that combines mirroring with redirection by deploying replicated data servers for file replication. Simple, on-demand file replication is used. Dynamic replica selection and server resolution are considered.

An interesting project for file replication within the I2-DSI project is presented in [29]. The focus is on data mirroring. Further work on web replication and optimal replica placement can be found in [99].

3.5.5 Discussion

The usage of databases is still a unique feature for a Data Grid. Let us compare this to the climate modelling community: in that research domain large collections of files are available and stored in so called "flat files" rather than databases. This requires additional data management tasks like keeping a catalogue of available files whereas in some physics experiments in the HEP community the database management system (DBMS) takes care of this.

Since recently, not only database research deals with data replication but also Grid research but tackles the problem from a different point of view. The aim of this section is to outline approaches in both research domains and to find commonalities in order to have a most efficient Data Grid that can manage Petabytes of data stored in object-oriented databases or object data stores. We will provide a first basis for such an effort and have published earlier results in [117]. Since Data Grids are very new in the research community, we see a clear need for identifying the characteristics and requirements of Data Grids and how they can be met in a most efficient way.

We first identify some selected related work in the database community as well in the Grid community. From this we derive commonalities and discuss briefly what the contribution for an efficient Data Grid can be.

Distributed DBMS Research

Distributed database research basically addresses the following issues:

- Replica synchronisation is based on relatively small transactions where a transaction consists of several read and/or write operations. In contrast, in HEP a typical data production job can write a whole file and thus a transaction should be relatively "large".
- Synchronous and asynchronous replication: Most of the evaluation techniques for performance of replication protocols are based on the amount of communication messages needed.
- Cost functions for network or server loads are rarely integrated into the replica selection process of a DBMS. In other words, statistics and performance information is hardly used for access optimisation to replicated data.
- Rather low amount of data as compared to the Petabyte scale of HEP.

Data access over the WAN and optimisation of replicas are rarely addressed in database research. In a DBMS there is normally only one internal method of accessing data. For instance, a data server serves pages to a client. In detail, a client sends a request for data to the DBMS, and the DBMS (in particular the data server) uses the implemented method for serving the requested data. For read and write access the same method is used, independent of the amount of data accessed. For a Data Grid, a single access method is not optimal depending on the number of objects to be accessed from a file. For accessing large amounts of data of a remote file, the usage of a file transfer mechanism to transfer the entire file and access the file locally can be more efficient in terms of response time than remotely accessing (and thus streaming) many single objects of a file. Based on cost functions it can be even more efficient to send the application program to the remote data [119].

Grid Research

The Globus project [41] provides tools for Grid computing like job submission, resource discovery, security, etc. Recently, Globus is also working on a Data Grid effort that enables fast and efficient file transfer, a replica catalogue for managing files and some more replica management functionality based on files. Replica update synchronisation is not addressed. In the Grid community there is a general tendency to deal with replication at the file level[2], i.e. a single file is the lowest granularity of replication. This has the advantage that the structure of the file does not need to be known by the replica manager that is responsible for replicating files from one site to others over the WAN. This is a valid simplification for most of HEP replication use cases and, thus, is also the main focus of this section.

Commonalities - The Usage of Replication Middleware

Research results can be combined from both communities by introducing a *replication middleware layer* that manages replication of files (and possibly some objects) by taking into account that each site in a Data Grid will manage data locally with a database management

[2]Note that Legion has a different approach and deals with objects rather than files.

system. Thus, the middleware is responsible for site to site (or *inter-site*) replication and synchronisation whereas the local DBMS takes care of transactions on local data.

In Grids there are many tools for monitoring applications and network parameters. These can be used for filling the gap. Hence, a hybrid solution between database and Grid research can be identified.

One of the main questions to be answered is to what level the replication middleware will be able to replace a pure distributed DBMS approach where also inter-site replication issues are dealt with. Clearly, with a replication middleware there are many more restrictions for update synchronisation and transparent access to data. There will also be a performance penalty for replica synchronisation. However, an aim of the replication middleware is to provide several relaxations of the concept of transparent data access and data consistency.

File Replication Issues

Although we have identified that file replication is less complex as regards the replication protocol, Data Grid environments place strict requirements on file replication tools which are often not dealt with in database systems. There are many more items that need to be considered than only update synchronisation. A major issue is high performance in data transfer and a need for a high-speed file transfer protocol. Thus, for a file replication middleware a major performance factor is fast point-to-point replication for rather larger files in the order of one Gigabyte. In addition, security is an important factor and all file transfer needs to be done respecting authentication and authorisation. All these issues are important requirements that we try to solve in the following chapters of this thesis.

Outlook and Potential Benefits of a Data Grid

We have discussed replication in several distributed systems and now motivate what the potential benefits and also drawbacks of a Data Grid are. One obvious feature of a Data Grid using middleware to deal with data management is that the middleware does not necessarily have full control over read and write operations on data. In other words, the Grid middleware has to rely on locking and transaction mechanisms on top of existing database or file systems. Also interfaces to data appear at different levels: POSIX file system interfaces versus database specific access methods (for instance through a database server) versus higher level programming interfaces on top of file systems or database systems.

A potential benefit of a Data Grid is that several sources of information exist to optimise either data transfer (fast file replication versus streaming of single objects over a wide-area network) or scheduling of jobs sent to a Grid[3]. Furthermore, there exist several Grid services that monitor performance information for networks and possible data servers. Although database management systems provide query optimisation functions too, they mostly do not take into account dynamic parameters like current network throughput or the load of data servers at remote sites. We see an additional benefit in using Grid middleware services to optimise data access over wide-area network connections. More details can be found in Chapter 8.

[3]In [20] we analyse economic models for efficiently scheduling jobs in a Grid.

Chapter 4

Architecture and Cost Model

In this chapter we propose a data management architecture for a Data Grid and mainly focus on file replication services which are discussed in detail in later chapters in this thesis. Once the architecture is in place, we introduce a new cost model for distributed and replicated data stores. This model can be used as a base for evaluating parameters and influences in a Data Grid, especially in the High Energy Physics environment. A novel object location table for locating replicated objects in a large Grid environment is introduced and we use our cost model as an example for the evaluation of the access to replicated data in a Data Grid.

4.1 Data Management Architecture

In Section 3.4 the DataGrid project was explained and the different work packages were outlined. In this section, we discuss in some detail the Data Management work package and propose a data management architecture. In our data management team, we have published early architectural proposals in [64]. Since the project evolved very much since our first proposal in the year 2000, a detailed, updated architecture for the Data Management work package can be found in [65].

The main objectives for data management are to compare and implement different distributed data management approaches including optimised data access, caching, file replication and file migration. We believe that such middleware is critical for the success of a heterogeneous Data Grid since it relys on efficient, uniform and transparent access methods to different kinds of data and storage systems. The main tasks can be identified as follows:

- management of a universal namespace for files (using replica catalogues)
- secure and efficient data transfer between sites
- synchronisation of remote copies
- (optimised) wide-area data access/caching
- management of meta-data like indices and file meta-data
- interface to mass storage systems

Architectural proposals and software solutions have to be integrated into the architecture of the entire DataGrid project as defined in [45]. A main integration criterion is that services and interfaces are well defined and usable for other middleware software as well as potential end-users.

The data management architecture and the corresponding software have to satisfy the stringent HEP data management challenges as pointed out in Section 3.3.3. As regards use cases in the Earth Observation domain, data is collected at distributed stations and are also maintained in geographically distributed databases. In molecular biology and genetics research a large number of independent databases is used which need to be integrated in one logical system [64].

4.1.1 Grid Services for Replication

We now discuss the main services and building blocks in the Data Management architecture with respect to replication. For completeness, a definition using DataGrid terminology is given. Since data is distributed and replicated to large sites using disk and tape storage, a storage element is the main resource where data (in principle files) are stored.

Definition: A *Storage Element (SE)* is a basic storage resource in a Data Grid and also defines the smallest granularity for a compound storage system. A Storage Element can either be a large disk pool or a Mass Storage System (MSS) having its own internal disk pool. A Storage Element is uniquely identified by host and domain name. Within a site (connected to other sites through a wide-are network link), several independent Storage Elements can co-exist.

Our proposed architecture includes the following main services: Replica Manager, Replica Catalogue, File Copier and Consistency Service[1]. The interactions of services and components used by the main services are depicted in Figure 4.1.

Shaded boxes represent services provided by WP2 and the transparent box (Replica Optimiser) represents a component that is used internally by WP2 services. Each of the services provides well defined APIs that either can be used by other services or end-users. A component on the other hand is used by a service and only provides an internal interface to the specific service. We start our discussion with the *Replica Manager* since it is the central service for a file replication process. The entire architecture discussed in this section uses files as the granularity for data management.

Replica Manager

The Replica Manager manages file transfers (replication) by using the File Copier service and replica catalogue information through the Replica Catalogue service. Arrows from one service to another indicate that the service is using another service. In Chapter 5 we present GDMP (Grid Data Mirroring Package) which is a file replication software tool that implements most of the Replica Manager functionality. The main tasks of the Replica Manager are to copy files securely and efficiently between two Storage Elements and update the replica catalogue when a file has arrived error free at the destination. More details on secure and efficient file replication for read-only files can be found in Chapter 5.

Replica Optimiser

Access to replicated files shall be optimised by using performance information like current network throughput and latency as well as loads on Storage Elements. The *Replica Optimiser* module - used by the Replica Manager - selects "best" replicas. More details on access optimisation is given in Chapter 8. Through access pattern management, i.e. monitoring read

[1] In [65] another two data management services - ServiceIndex and SQLDatabase service - are described and we refer to that document for further information.

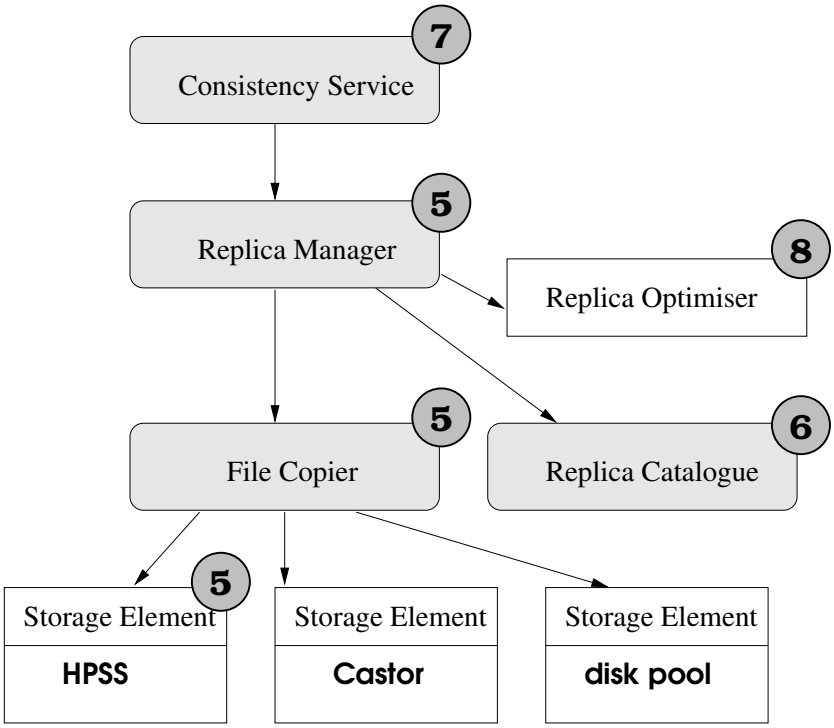


Figure 4.1: Interaction of main services and components. Numbers indicate the chapter where details are provided.

and write access to replicated data, for a given query an optimal migration and replication execution plan can be produced.

An additional task for the Replica Optimiser is to do query estimation, i.e. estimate how long it takes to serve a query asking for a certain amount of data. Since large data stores are used, certain access control is required to limit the amount of data to be queried and thus transferred to the client application.

File Copier

The *File Copier* (also called *Data Mover* in Chapter 5) is an efficient and secure file transfer service and has to be available on each Storage Element. File transfer takes place between two Storage Elements. In Chapter 5 we elaborate on FTP performance tests.

Storage Element

Data Management in a Data Grid has to deal with a heterogeneity of storage systems and thus Storage Elements. The interface to the Storage Element has to be unique regardless of the underlying storage technology. Possible storage system implementations include mass storage systems like HPSS [67], Castor [8] or Enstore [36] as well as distributed files systems like AFS [66] or NFS [109]. All these implementations can potentially co-exist and the Storage Element subsystem has to hide from higher layers the complexities and specific mechanisms for data access which are particular to every storage system.

Replica Catalogue

As defined in Chapter 2, the *Replica Catalogue* provides the mapping between logical and physical filenames and allows for updating and querying replica information. Basic architectural considerations are given in the next section and a distributed replica catalogue system is proposed in Chapter 6.

Consistency Service

Since not all data is read-only and some files need to be updated and kept in sync with other replicas, a *Consistency Service* is provided on top of the Replica Manager. The task of the Consistency Service is to implement several replication policies and update distributed and replicated data stores. More details on replica update synchronisation are given in Chapter 7.

Security

All services and components need to provide appropriate security mechanisms that transparently span world-wide independent organisational institutions. In general, global and local security aspects have to be considered. The site owning data has to make sure that the remote sites provide the same level of security as the owner requires for her data. This will be a serious issue when dealing with sensitive data where human or intellectual property rights exist [64]. Consequently, interactions between services across network connections have to be done in a secure way. Only authenticated and authorised requests are allowed.

As an example for possible security problems we point out synchronous replication strategies that require indefinite write permissions to remote Storage Elements when file updates are propagated. On the other hand, on-demand replication where updates are propagated at certain times, have less security concerns for participating sites.

4.1.2 Replica Catalogue Design

Access to data is most important for data analysis in the HEP as well as in any other scientific community. Access to replicated data requires specific data and meta-data structures and can be addressed in different ways. A database management system (DBMS) has an internal access method whereas in a Grid environment these data structures are provided externally as a service to the end user. Here, emphasis is put on how to combine techniques from a Data Grid and a database environment.

The management of replicas requires certain meta data structures that store information about the distribution of and access to data. Whereas an object location table [63] can be the data structure for managing objects, for files we can use a file replica catalogue. Recent database research does not address replica catalogues at all and thus their approaches are not published. On the other hand, in the "young" Grid research community, file replica catalogues are essential and an active research area.

Architectural Details

Before we go into details of replica catalogue usage, we briefly discuss architectural aspects and details like which information needs to be stored in a replica catalogue and what query functionality should be provided. We focus on replica catalogues holding references to files rather than objects.

In principle, a replica catalogue stores a mapping of logical filenames to one or more physical filenames. Whereas a logical filename has to be location independent, a physical filename provides information where a file is physically stored. Thus, the most simple replica catalogue

is a table with two columns: one holding logical filenames and the other one storing physical filenames (including file locations). We now assume that files are stored on disks and disk locations do not change. We use this simplified approach first to illustrate a replica catalogue example. In a more realistic Data Grid scenario where mass storage systems are used, physical files are not always available on disk but also on tape locations.

Table 4.1 gives an example where two logical files *file1* and *file2* are partially replicated to three Storage Elements. Note that we do not impose any naming conventions here for logical filenames but do so for physical filenames. A physical file is always stored on a particular Storage Element which is uniquely identified by host and domain name.

logical filename	physical filename
file1	host1.cern.ch/data/file1
file1	server22.slac.stanford.edu/run1/data/file1
file2	host1.cern.ch/data/file2
file2	server22.slac.stanford.edu/run1/data/file2
file2	host37.infn.it/data/file2

Table 4.1: Two logical files partially replicated to three Storage Elements (SE). The physical filenames contain the SEs.

Definition: A disk resident *physical filename* contains the host name and domain name of a Storage Element where the file is stored plus the entire file path within the Storage Element. Note that the Storage Element here can only be a disk server or disk pool rather than a Mass Storage System.

Except the basic two entries (logical filename and physical filename), a replica catalogue can contain additional information (attributes) for logical as well as physical filenames. Since all physical file replicas are identical in size, logical file attributes like size, time stamp (creation date), checksums, file type etc. can be added. A typical physical file attribute indicates if a replica is a master (possible updates or modifications on the file can be done) or not. Consequently, a simple table as illustrated in Table 4.1 does not contain all required information and a more advanced schema for storing catalogue data is required. More on catalogue schema is given in Chapter 6.

Definition: A replica catalogue is responsible for managing the *name space* of logical and physical files, i.e. it has to guarantee that filenames that are used in a Data Grid are unique within a particular name space.

Since a replica catalogue is a Grid service, it needs to be accessible for applications or other Grid services over the wide-area network. Thus, a network protocol is required for accessing/querying catalogue information. In general, Grid services rely on established Internet protocols. The Globus replica catalogue relies on the Lightweight Directory Access Protocol (LDAP) [136] for access to the replica catalogue. LDAP servers thus serve lookup, insert and delete requests. More information on the Globus Replica catalogue implementation can be found in Chapter 6. In the DataGrid project, HTTP [38] is considered to be the protocol used for catalogue interaction.

Replica catalogue information - like any other file meta-data - needs to be stored persistently in a storage system. Most conventional and efficient storage systems are databases. In other words, a database backend is required which provides fast access to concurrent clients

and allows for concurrency control in order to avoid read-write anomalies with concurrent readers and writers to the same data. A conventional database management system supporting database transactions is a viable solution for a database backend. For both, LDAP and HTTP front-ends, database back-ends can be used. In case of LDAP, some special propose LDAP servers are available that also provide special purpose database back-ends whereas for an HTTP approach, only conventional databases (commercial, open source, object-orient or relational) can be used. Note that although LDAP provides add and delete functionality, transactions and concurrency control is not provided and needs to be taken care of by the underlying database back-end.

Usage of a Replica Catalogue

Independent of the architecture of the replica catalogue and the underlying storage back-end for replica catalogue information, we now discuss how a file replica catalogue can be used together with a database management system. We are now assuming that actual data (e.g. event data in High Energy Physics) is stored in a database management system like Objectivity/DB or Oracle.

In principle, a DBMS provides, by definition, a catalogue of files that are available. As for Objectivity/DB this is an explicit file catalogue. Basically, a replica catalogue does the same for arbitrary files (of any file type) which may or may not have connections (associations between objects in two or more files) to each other. Now the question arises what is the use of a replica catalogue when data is stored in a DBMS? In order to answer this question, we identify the key problems that need to be addressed in a heterogeneous Data Grid environment with many sources of data: managing replicas of files and dealing with heterogeneous data stores.

- **Managing replicas**: As for Objectivity, a file catalogue exists. The granularity for replication is now assumed to be on the file level. When multiple copies of a file exist, they have to be introduced to a global replica catalogue which is an extended version of the native file catalogue of the DBMS. Furthermore, a name space has to be provided that takes into account multiple replicas. Both features are enhancements to a DBMS that only considers single sources of information. Note that we assume here that the underlying DBMS does not support file replication. A generic replica catalogue with a well defined access protocol and interfaces can be used. Another approach is to simply build the catalogue and the access methods with the native schema of the DBMS and store the replica information in the same DBMS where the actual data is stored.

- **Heterogeneous data stores**: The diversity of the HEP community requires an approach which supports heterogeneous data stores since different kinds of data may be stored in different formats and also in data stores of different vendors. Even the two different database paradigms, relational and object-oriented, should be supported. In this case, using a general replica catalogue seems to be a good choice since replica location information can be accessed in a uniform way without the knowledge of the underlying data store.

Combining a DBMS (storing real data) with a replica catalogue service (storing replica information and providing the corresponding access methods) means to expose the replica catalogue to a larger user community, where it is not necessary to have a database specific front end to access the replica catalogue. The usage of a replica catalogue allows an open environment (open for several, heterogeneous database management systems for storing data) and hence allows the integration of data sources of different kinds. Note that the problem of accessing heterogeneous data sets is not addressed here and may require mediators [129]

for data integration. However, introducing a replica catalogue is at least a starting point for extensibility.

We conclude that the usage of a replica catalogue service in combination with a DBMS seems to be useful in a heterogeneous environment and for synchronisation of sites holding replica catalogue information.

4.2 A Cost Model for Data Distribution and Replication

Since Data Grids rely on Internet protocols and technology, one can take it as an analogy to illustrate the expanding possibilities of Data Grids. With the growth of the Internet in the last couple of years and expanding technologies in database research, data warehousing, networking and data storage, large distributed data stores with data amounts in the range of Petabytes are emerging [76]. Not only the choice of the optimal data storage system (relational or object-oriented databases, flat files, etc.) is essential but also the placement of certain parts of the data store has a major influence on the response time of the entire system. Hence, an analysis of costs and influences of replicated data stores is required which helps to solve the data placement optimisation problem.

In this section we provide a novel cost model we proposed in [119] for building a world-wide distributed Petabyte data store. We will elaborate on a framework for assessing potential system costs and influences which are essential for the design of the data store. In the proposed model, the *response time* for serving multiple user requests is a major performance factor of a data store. We require a high-throughput rather than a high performance system which just serves a single user per time. High throughput requires several factors to be considered and optimised (see Section 3.2 for a longer discussion on high throughput versus high performance).

As illustrated in Chapter 2, in the database community, replication research mostly deals with update synchronisation of synchronous or asynchronous replicas. Read or write transactions are normally rather small and hence a small transfer of data is done over the network. In particular, [84] proposes a performance model for telecommunication applications which have the previously stated features regarding transactions. The main assertion is that a fully replicated data store is suboptimal since the update propagation to each replica takes a long time and hence decreases the response time for write transactions. However, partially replicated data stores do not provide optimal read response times since only parts of the data items are replicated. The trade-off between optimising the response time for read and write transactions by considering highly consistent replicas has to be found.

We limit our problem domain as compared to [84] and address large scientific data sources and computing environments that mostly deal with read-only data like CERN's LHC experiments. Furthermore, transactions are very large and write data of about one Gigabyte within a single transaction. Consistency of data is not as high as in a real-time distributed database management systems. This imposes different requirements which have to be addressed. We analyse influences and costs of very large, replicated data stores over the wide-area network which are not addressed in database research. These can be directly used in the design of such data stores. Much emphasis is put on the placement of data items and the influence on the response time of the data store. A detailed cost model and some tradeoffs are given.

4.2.1 Model Assumptions

For the cost model we have the following input parameters: Physicists are working at distributed locations in the world requesting access to the data store (see Figure 3.4). We further assume a client-server model where a data store has one or more *data servers*. The time which elapses between sending the request and retrieving the required data locally is defined as the

response time of the system. The location of the client application has an important impact on the response time because a LAN normally has a much higher bandwidth than a WAN. This results in a shorter response time which we consider as the major performance measurement for the data store.

The response time for read-only data is optimal when all data is available at each site. However, full replication comes at higher cost for infrastructure. Thus, if the costs of storing replicated data and the response time still should be kept low, certain queries and their impact on data access have to be considered.

4.2.2 Location of the Data Store

In general, data can either be located at a centralised, single data store or distributed and replicated to several data stores. A brief discussion on both possibilities is given below.

Centralised Data Store

Given a centralised data store with a single data server, all the incoming requests from application clients have to be served by the centralised data store. All users have to send their requests to the single data server regardless of whether the user runs the application client over a LAN or a WAN. The consequence is that the response time for WAN users will mostly be considerably higher than for LAN users. For a large number of concurrent users a single data server can easily become a bottleneck for a system with a centralised data store.

This model is in contrast to a distributed Data Grid.

Distributed and Fully Replicated Data Store

Based on the fact that many physicists will do their analysis work at different sites in a Data Grid, we can identify a distribution of users at certain centres, an inherent distribution of application clients and hence also a distribution and replication of data. According to the principle of *data locality* we do not want that users access data via the WAN. Instead, data should be located such that users can access data in the same LAN.

The availability of several data servers and multiple copies of data items result in a relatively small load for each single data server. Thus, there is a reduction of the bottleneck of a single, central data server, because of the distribution of requests to several data servers. In terms of user response time it is certainly better to provide multiple copies of data at several distributed sites in order to have data as close as possible to the user.

Based on the model assumption and the distribution of users, the distributed data store is preferable for the LHC computing model and thus for the European DataGrid project.

4.2.3 Costs and Influences of the Data Store

We now have a more detailed look at data locality and how this can influence the response time of a system. We will distinguish between *system influences* on the response time (static and dynamic ones) and *economically relevant costs* that appear in distributed systems and discuss additional sources of influence presented in [119].

Static System Influences

Static system influences are supposed to be stable and do not change over time. In general, these influences are easy to measure compared to the dynamic system influences.

Bandwidth of the network link: The physical network connections are of major importance for a WAN. However, since we are referring here to static system influences, the current network load is not considered here but falls into the discussion of dynamic system influences.

Size of data to be transferred: The size of data to be transferred has an influence on the time it takes to transfer data from one point to another one. There is a direct proportional relationship between the transfer time and the data size. Based on the amount of data to be transferred and the time constraint when the data has to arrive, the network can be a bottleneck. When the data size increases, not only the storage capacity (which grows as the amount of replicas grows) is increased, but also the the network bandwidth utilisation grows since more data has to be transferred over the network when data items are replicated. Based on the data requirement per time it is important to determine if it is possible at all to transfer the required data amounts over the network link. An alternative is to transfer parts of the requested data by normal *"air mail"* via magnetic tape.

Latency of a network link: This is can be measured by the *round trip time (RTT)* of a single data packet which tells how long it takes to send a single data packet over the network.

Network protocol: Although the latency is an indicator for the distance in the network, the deployed network protocol (TCP, UDP, FTP, ...) influences the response time very much. If a protocol implements many acknowledgements, the RTT delays the overall time to send the requested data. Hence, the selection of a reliable protocol with many acknowledgements has to be compared to an unreliable protocol with less or no acknowledgements.

Dynamic System Influences

These are influences that change dynamically and require a constant monitoring system.

Current throughput - network load: In a multi-user environment there is a high factor of uncertainty for the currently available throughput for a given time. This is very much dependent on external influences and also on peak times (for instance users in different time zones) when the network is utilised heavily. Bandwidth reservation and the usage of dedicated lines can significantly contribute to reduce the factor of uncertainty for the throughput. However, this comes again at an additional cost and network bandwidth reservation can only be used for particular network technologies e.g. ATM networks.

Current "data server load" and disk speed: Not only a single data server can be a bottleneck that reduces the response time, but even the speed of the disk subsystem can have a negative impact on the performance of the system. We also have to take into account the number of jobs waiting to be served by the data server.

Application Dependent Influences

Access patterns: Replication can be regarded as a form of caching which reduces the amount of data traffic over the network. The *access patterns* of the users and hence the amount of data to be requested within a particular amount of time have to be considered. When we assume that we have n sites and therefore n times as much as data in the entire system (we assume a fully replicated system with a replica of each data item at each site), one data item has to be requested on average n times in order to profit in terms of network traffic reduction from a distributed and replicated system.

Data consistency: The *data consistency factor* is the time delay with which copies of data or updates arrive at a site and a fully equivalent copy is created. The lower the factor, the less up-to-date is a remote site. This factor, also known as *relaxed coherency* [43], has to be agreed on by the database administrators and the users, and is also very much dependent on the available network bandwidth. In other words, the user has to tell what amount of consistency is required and this has to be achieved by possible replication mechanisms. For instance, we can identify rather loose consistency where update synchronisation takes place every minute, hour or even day. If strict consistency is required, replica update and synchronisation is done in real time and immediately after changes have been made.

In addition, distributed data stores have to exchange communication messages for (meta) data updates. Even if all data is read-only, it has to be introduced to each distributed data store at least once. Since multiple updates can occur at the same time in a distributed, multi-user system, global transactions have to ensure data consistency at multiple sites [49].

Degree of replication: Whereas in a fully replicated system each data item is replicated to 100%, in a partially replicated system certain data items can occur only once while others occur multiple times. The degree of replication represents the percentage of how often a data item is replicated. The exact placement of a data item and the amount of replicas for each data item have to be optimised and highly depends on the access patterns and the data consistency factor.

Quantity and quality of data: Quantity and quality of data can represent different costs for a data location cost model. Even if a large quantity of data is frequently requested, it does not necessarily mean that this data always has to be replicated, e.g. there may be a smaller set of data that in relation is not as frequently requested as others but has a higher quality and thus higher priority for the final result of the query and/or the computation. For example, data that contains the Higgs boson[2] appears not very frequently (low quantity) but is very important for physics analysis (high quality) [107]. In a partially replicated system the choice of which data to be replicated has to be based on these quantity and quality constraints. A higher quality/quantity of data corresponds to a higher possibility of replicating data to several sites.

[17] points out that the main problems of generic replication solutions is that knowledge about the data is required. In order to have a correct quantity/quality tradeoff, the user input and/or some hints are required for deciding about the creation and allocation of replicas. Let us define the choice of having a certain amount of a particular set of data items the *replica selection* process.

Replica placement: The I/O requirements of users should be served as fast as possible. If most of the time of an application is spent on I/O, the requirement to replicate the requested data increases. In contrast, the need for replication decreases when very much time of the applications is spent on CPU (i.e. computation) and only small amounts of data are requested. *Application monitoring* which keeps track of user access patterns and the time spent on computation have an important influence on the optimal degree and placement of replicas.

An advanced, partially replicated system can automatically and dynamically create and delete replicas based on access patterns in order to have better response times (by creating replicas) and reduced storage space (by deleting not very frequently requested replicas). A more detailed discussion on automatic replication adapted to access patterns of users can be found in [133].

[2]The Higgs boson is a sub atomic particle which is the key for the origin of matter. The primary task of LHC experiments is to show by experiments the existence of the Higgs boson.

Data compression: The *compression time factor* has a negative influence on the data consistency factor since additional computing power and thus time delay will be required but this can be evened out to a large extent by the *compression factor* which results in less network bandwidth utilisation. The tradeoff between these two factors has to be found out. By an increase of computing power over time which is predicted by Moore's law [105], the compression factor becomes more and more important.

Economic Costs of a Distributed System

The advantage gained by data locality comes at the costs of other factors and constraints that have to be analysed. In particular, multiple copies of data result in more resources and more sites to be maintained. We define a *site* as a number of computing resources (CPUs) and data storage devices connected via a network link to another site. In the CERN environment a site is either CERN or a Regional Centre. The following additional costs and constraints occur in a model with multiple sites:

Infrastructure costs: These costs cover storage devices like disks and tapes, CPU and network resources that have to be put in place. Additionally, user support and maintenance of each single site, and regular replacements of devices have to be taken into account. Maintenance also covers costs for hardware and software personnel. One might claim that a distributed system in general requires network connections to remote users. We can clearly see from our application that only the distribution of users causes already additional network costs.

Policies and political influences: Finally, political considerations which constrain technically possible solutions have to be added to the technical ones mentioned above. A political constraint can be that a site only wants to be updated every 5 hours. This has to be expressed by the data consistency factor. Another example is that a site wants to get only a particular set of files and partially replicated data stores are the consequence.

4.2.4 The Cost Model

The target function of a cost model for a distributed and replicated data store is the response time of a user's request, which is the focus of the minimisation process.

In the previous section, we listed different influencing factors for our cost model and now express these factors with a mathematical notation for further analysis. Basically, we have to distinguish two different approaches for the resolution of a user's request, *data migration* and *application migration*. The model can also be found in [119].

Data migration. In this approach, the data moves across the network and the application executes locally. The response time of a user's request $Time_{data_mig}$ is the sum of 3 terms: the time for accessing local data of size D_{local}, the time for accessing remote data of size D_{remote}, plus the time $Time_{dist}$ to cover the latency to a remote location (which is zero in the special case if everything is accessed only locally). Each term is dependent on the replication factor, the pattern of the system and the access pattern of the user request. We assume for simplicity that the access pattern is equally distributed as the replication pattern, which allows us to express the replication simply by a replication factor rep as a fraction (from 0 to 1). Given the bandwidth for the LAN by BW_{LAN} and for the WAN by BW_{WAN} we define the following general formula

$$Time_{data_mig} = \frac{rep \cdot D_{local}}{f_{LAN} \cdot BW_{LAN}} + \frac{f_{D_{remote}} \cdot (1 - rep) \cdot D_{remote}}{f_{WAN} \cdot BW_{WAN}} + Time_{dist} \qquad (4.1)$$

In this formula three influencing factors, f_{LAN}, f_{WAN} and $f_{D_{remote}}$, describe the static and dynamic system situation.

The bandwidth of the LAN is influenced by the server load $sl(t)$, a time t dependent value between 0 and 1 (0 = no influencing server load, 1 = server fully loaded) and the network protocol np (0 = no network protocol, only data transfer; 1 = only protocol, no data transfer). This leads to

$$f_{LAN} = (1 - sl(t)) \cdot (1 - np) \tag{4.2}$$

The bandwidth of the WAN is influenced by the network protocol too and, instead of the server load, by the network load $nl(t)$ (0 = no additional network load; 1 = fully loaded network, no free bandwidth). Thus

$$f_{WAN} = (1 - nl(t)) \cdot (1 - np) \tag{4.3}$$

The data size transferred from a remote site can be influenced by a data compression factor $dc(D)$ dependent on the data D, where 0 denotes no compression possible; 1 (theoretically) full compression, no data needs to be transferred. We can therefore define

$$f_{D_{remote}} = (1 - dc(D_{remote})) \tag{4.4}$$

The other named issues influencing the behaviour of the data store can be interpreted as restrictions, e.g. the relation between disk (bandwidth of the disk system BW_{disk}) and network characteristics (BW_{LAN}, BW_{WAN}) expressed by $BW_{disk} > BW_{WAN} > BW_{LAN}$, are beyond the decision of this model, e.g. policies and cost limits.

Application migration. Another approach for the request resolution is to migrate the application to a remote data store, execute the application there and send the result data to the requesting client. The time for this way of execution summaries four factors: the time for sending the application D_{app}, the time for accessing the requested data locally at the remote data store, the time for sending the result D_{result} to the client and finally the time to cover the latency between client and server. This leads to the formula

$$Time_{app_mig} = \frac{D_{app} \cdot f_{D_{remote}}}{f_{WAN} \cdot BW_{WAN}} + \frac{D_{local}}{f_{LAN} \cdot BW_{LAN}} + \frac{f_{D_{remote}} \cdot D_{result}}{f_{WAN} \cdot BW_{WAN}} + Time_{dist} \tag{4.5}$$

The influencing factors are the same as in the formula for $Time_{data_mig}$. The replication factor is not applicable due to accessing the data fully at the remote site.

Target function of the cost model. To sum up, the target function for the minimisation of the proposed cost model is the minimum of both possible approaches, which can be expressed finally by

$$Time_{request} = Minimum(Time_{data_mig}, Time_{app_mig}) \tag{4.6}$$

Discussion of tradeoffs concerning the CMS experiment: Assumptions and information gained through our internal studies, Monarc [80] and RD45 [101], revealed that the raw data of High Energy Physics (HEP) is in general very difficult to compress. What is more, the LAN links are expected to getting cheaper faster than the CPU gets faster. Both factors are strongly in favour of investigating more effort into higher LAN network connections rather than into research of efficient compression algorithms. However, as soon as data is transferred over

the WAN, data compression becomes a more important issue since, for instance, meta-data for global access and transaction control are assumed to be much more easy compressible. In addition, WAN connections are behind the capacities of CPUs and disk subsystems assuming a maximum of about 155 Mbps network links in practise. This is of course not true for potential Gigabit links as demonstrated at super computing conferences in 2000 and 2001 [124].

A further improvement of the throughput can be gained by the inherent relaxed consistency. In other words, the consistency of the replicas in the distributed system is not such an important issue in HEP and, for instance, updates of newly created physics algorithms do not need to be propagated in real-time. Thus, an increased replication factor yields higher data availability in case of a network failure and a higher read-performance due to local data placement.

4.2.5 Related Work on Cost Models

Related work on performance and cost models can be found in performance modelling of distributed systems and distributed database systems. However, a fully integrated cost model which covers a distributed data store (Data Grid) as well as distributed system features is not discussed. Data servers can be modelled as queueing networks [85] where single transactions are analysed and represented by their arrival rate at the data server. Large data sizes are not mentioned.

A very detailed analytical queueing model can be found in [84] which uses telecommunication applications for replication performance models. An important feature of this work is the identification of throughput, bottlenecks and the correct placement of replicas. Access patterns and workloads are modelled by arrival times of transactions. A detailed study on replicating some objects to some sites is presented. Update transactions happen rather frequently. Therefore, the degree of replication can have a negative influence on the response time of the system since there is an increasing update overhead as the amount of replicas is increased. Further file replication models can be found in [68].

4.3 Example

In this section we propose an Object Location Table (OLT) which is used for locating single objects in a Data Grid and providing efficient access to the data. The architecture of a particular object location table that exploits the data model of High Energy Physics experiments, in particular CMS, is presented. A main focus is to use the our presented cost model and analyse the data management problem by example. Here, we consider object level replication and a cataloguing mechanism for replicated objects.

Definition: An *object location table* is a replica catalogue for objects in a file.

4.3.1 Introduction

As we outlined in Chapter 3, Data Grids and in particular HEP experiments deal with the problem of locating objects distributed world-wide in a Data Grid. Local or remote replicas of individual physics objects rather than entire files have to be provided. A distinction between logical and physical objects is required where a logical, location independent object descriptor is resolved into a physical location where an object replica can be found. The object location table can be seen as a special data structure to support this mapping. In the remainder of the section we use the term *object location table (OLT)* in order to define a replica catalogue for objects.

We have proposed a first architectural model and a specific implementation of the object location table in [60] and state the main ideas here. We argue that such a table can be

efficiently implemented by limiting the freedom of lookup operations, and by exploiting some specific properties of the physics data model. Thus, the proposed table is rather specific for the HEP data model and needs to scale to at least some 10^{10} objects.

As pointed out in Section 3.3, the CMS experiment will store more than a Petabyte of data in a database management system a year (10^9 events a year) starting from 2006. The database management system can have an object-relational or pure object-orient API. In both cases, single objects are addressed via unique object identifiers (OIDs). Given that a physics event will have at least some 10 objects, at least 10^{10} objects are created annually. Each of these objects can potentially be replicated to a site in the Data Grid. Multiple replicas of an object can also exist at a local site when objects are reclustered [61] or staged from tape to disk [60]. In both of these cases, an object location table (OLT) is required to resolve the logical, location independent object descriptor into a physical location where an object replica can be found.

Similar to the definition of logical and physical names for replicas, single objects need to be named and uniquely identified by an OID which can be logical or physical. Logical OIDs are preferred when data movement comes into play[34]: since there are no permanent addresses stored in the OIDs, the objects can be moved freely. Additionally, object replication and access to different versions can be supported effectively. The price for this flexibility is that the OID has to be mapped in a lookup step to the permanent address of the object. The object location table presented here uses logical OIDs.

4.3.2 The Minimisation Problem

The aim of the OLT is to enable and optimise access to a set of physics objects which may be partially replicated [60]. The location where the job has to run is already determined and thus it needs to request data from remote sites. The OLT not only provides lookup operations, but also determines which replica to return in any lookup operation in order to minimise the access cost. In other words, the OLT provides some replica access optimisation functionality (see Chapter 8 for a more detailed discussion). In particular, we minimise equation 4.6 of our cost model presented in Section 4.2.4. Since the OLT does not do any job migration but rather finds out which data is optimal to transfer, we can limit our minimisation problem to data migration only and thus to equation 4.1 in Section 4.2.4.

In addition, the OLT can also be used for replicated/distributed object within a site: for mass storage systems which stage (replicate) from tape to disk, and for systems that replicate objects while reclustering them, like [58] and [59].

The OLT system exploits a useful property of physics analysis to ensure that many lookups can be done for every block of index data read from the disk [60]. In physics analysis, at least for CMS, the order in which the individual events are processed is irrelevant. Furthermore, it is preferred to process events in time order (i.e. in the order in which they occurred in the detector), because this optimises the access to time-dependent calibration data. We transform this preference into a requirement: the OLT system requires that, at least on the small scale, events are iterated over in time order. This implies that individual objects are always accessed, and thus looked up, in the time order of their corresponding events. By storing location pointer values in the same time order, we can process many location lookup operations for every single disk block read.

4.3.3 The Logical OID and the Job Model

We claim that the traversal and the usage of the OLT depends on the job model and thus use the definition of [60] to explain briefly the logical OID and the job model. A job in the physics environment can be any executable that uses some input data and produces some output data.

In the proposed OLT system, a logical OID is a unique identifier of a particular object that holds physics data for a particular event. The OID has the following three fields [60]:

(*chunk ID*, *event sequence number*, *type/version ID*)

- *chunk ID* and *event sequence number* uniquely identify the event to which the object belongs. A *chunk* can be regarded as a collection of objects defined by the software framework that is responsible for writing (creating) physics objects. The event sequence number is simply the position of the event in the chunk to which it belongs.
- *type/version ID* uniquely identifies the object among the different objects that can exist in the system for that event.

For the OLT, we define a physics job as follows and every job specifies [60]:

1. An event set S of the events which must be visited, i.e. requested as input data to the job. We assume that each event in S is represented by its (*chunk ID*, *event sequence number*) pair.
2. A *job function*, a piece of executable code (for example in a loadable shared library), to be run for each event.
3. the type/version ID for every different kind of object to be requested
4. (Optional) If - for some type/version ID t- the corresponding object is not needed for every event in S, a smaller event set $S_t \subset S$ can be specified, with the understanding that the job function will need the objects t only for the events in S_t.

Before a job is executed at a predetermined site, the system computes the set C of all chunks that contain one or more objects identified in S. For every chunk in C, the system runs a *sub-job* over the requested events in this chunk. The sub-job visits the events in the order of increasing event sequence number, passing the requested objects to the job function.

4.3.4 Architecture of the Object Location Table

The OLT consists of several index levels that are part of the architecture. Figure 4.2 shows the physical representation of the entire object location table, and some stored physical objects. A particular example is illustrated where three different *collections* (a collection is a set of physical objects which are stored together) exist at two geographically distributed sites named CERN and RC1. The site CERN (left of the vertical dotted line) holds two collections: one on disk (with some physical objects shown in the left bottom corner) and the other one on tape (only pointers to the objects shown). While the collection on tape contains all possible objects of the corresponding chunk and type, the collection on disk only stores a subset of the objects. Another subset is stored in the collection on disk in RC1 (right of the vertical dotted line).

Let us now describe the different indexing levels in the OLT, from top to bottom. First note that the top four levels (above the horizontal dotted line) exist in two identical physical copies, one for each data location (CERN and RC1). This ensures fast read access to these parts of the OLT no matter whether a client that performs object lookups is running at CERN or RC1.

The top two index levels represent the type/version ID and the chunk ID of the corresponding object. Using these, a Top Level Collection Index (TLCI) can be obtained, which holds the list of all collections for a single chunk and type/version ID combination. A TLCI-pointer is a physical pointer to the Collection Contents Index (CCI). The CCI contains a list with the event

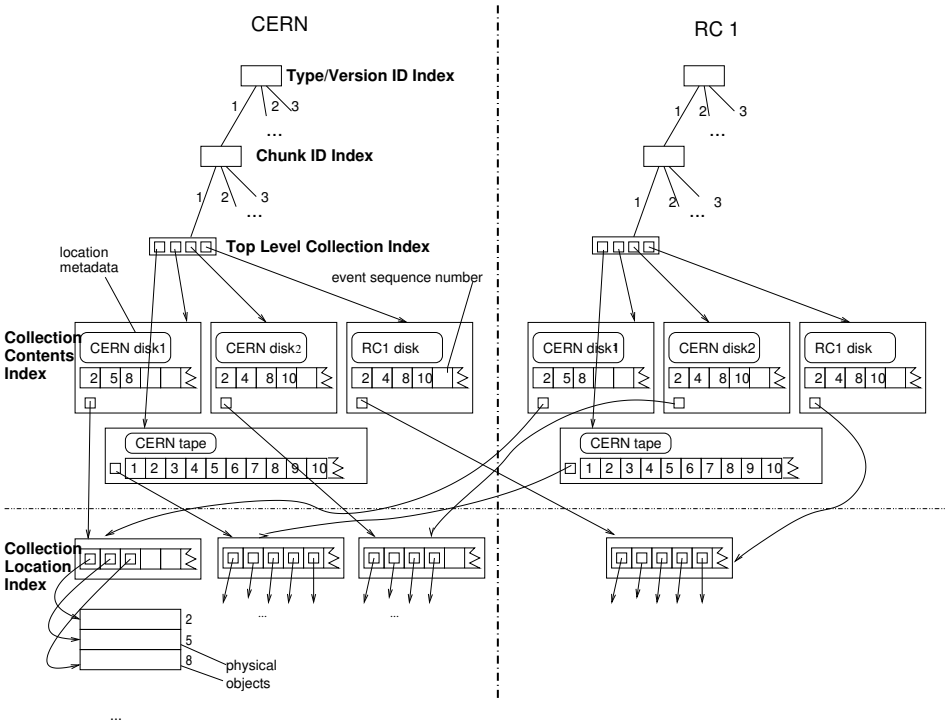


Figure 4.2: Physical representation of the whole object location table with its different levels of indexes, and some stored physical objects. Arrows indicate location dependent, physical pointers.

sequence numbers of all objects in the collection, and meta-data about the physical location of the collection. In the figure the meta-data "CERN disk" means that the collection is located on the disk pool at CERN. This meta information can be used to determine how costly it is to read the physical objects in collection. A pointer in the CCI further points to the Collection Location Index (CLI). The CLI resides physically at the location where the actual physical objects are stored. The CLI has pointers to each physical object in the collection. Event sequence numbers in the CCI and pointers in the CLI always appear in pairs, i.e. sequence number 3 in the CCI "CERN disk" corresponds to the first pointer in the corresponding CLI. Such a pair can be regarded as a mapping of the sequence number to a physical location of an object.

4.3.5 Use of the Object Location Table in Sub-job Execution

We now give a simple example on how the OLT can be used but refer to [60] and [63] for more details. The OLT automatically determines which replicated objects to access in order to minimise the access cost (replica access optimisation). This optimisation process happens at the level of the execution of a single sub-job. As an example to show the optimisation issues in executing a sub-job, we take a sub-job that should access the objects with some type/version ID 90 for four events in a set S_5 in chunk 5. As the chunk ID and type/version ID are fixed for the whole sub-job, we can represent the identity of the objects needed with the event sequence numbers in their logical OIDs only. Assume that the following objects are requested:

$O = \{2, 4, 5, 8\}$

Further assume that there are four collections relevant for the job. The contents of these

collections can again be represented as sets of event sequence numbers.

collection 1 = { 1,2,3,4,5,6,7,8,9,10 } located at CERN on tape
collection 2 = { 2, 5, 8 } located at CERN on disk1
collection 3 = { 2, 4, 8, 10 } located at CERN on disk2
collection 4 = { 2, 4, 8, 10 } located at RC1 on disk

Assume the (sub) job runs at RC1. The optimiser has to answer the following question: which replicas should be read by the sub-job to obtain the best overall performance for running at RC1? Based on the information of where objects are located, the answer of the optimiser looks like follows:

Read object 2 from collection 4
Read object 4 from collection 4
Read object 5 from collection 2
Read object 8 from collection 4

The OLT system including the optimiser automatically calculates the best answer at the start of the sub-job by comparing the set O of objects needed to the contents of all relevant collections. To make these comparisons, only the indexing structures shown in a topmost quadrant of Figure 4.2 are needed. The replicated objects themselves are only touched when the sub-job starts its iteration over the events.

Like in a general optimisation model, a *cost function* is used for each collection in question, to calculate the costs of reading a specific number of objects from that collection. Collections at different locations (CERN or RC1), or with different access characteristics (disk or tape), have different cost functions. The optimisation process can be found in [63]. Results are also based on earlier work in [59, 58].

4.3.6 Applying the Cost Model

We now use our cost model as the input cost function for determining the optimal objects to access. Since equation 4.1 consists of three main terms (access data locally, access data remotely and time distance), we first need to find out where the requested objects are located. Based on our object set $O = \{2, 4, 5, 8\}$, objects 2, 4 and 8 are locally available whereas object 5 is only available remotely and the access to this object needs to be optimised. In general, since all objects need to be accessed locally once they are available at the local data store, for remote objects the second and third term of equation 4.1 need to be taken into account whereas they can be neglected for objects that are already available locally. Consequently, we now only need to determine from where to transfer the remote object 5.

In our example, object 5 is replicated at CERN disk1 and CERN tape. We assume that the object size is 1MB and now need to determine the access time for remote access. Table 4.2 lists the parameters for local and remote access at time t for RC1 disk, CERN disk1 and CERN tape. We assume a replication factor of 50 percent and also a network protocol factor of 50 percent for each site. For remote access, no data compression is used.

The table shows the latency for accessing the object locally and remotely. We now need to compare the remote access times for accessing object 5 from CERN disk1 and CERN tape plus the time distance since these are the dominate terms for choosing a replica. The remote access time for object 5 at CERN disk is given by (0.5/(0.15*10). If we now add the time distance parameter, the total remote access time is 0.43. Compared to CERN disk1, for CERN tape we get: (0.5/(0.125*10) + 30 = 30.4. Thus, the minimal value is clearly 0.43 and thus disk1 is the location where object 5 is copied from. Thus, the decision of the scheduler from the previous section is justified.

In the cost model presented here the server load is only included explicitly in the local access. For remote access, it can also be included into the factor $Time_{dist}$ which is the case

parameter	RC1 disk	CERN disk1	CERN tape
$D_{local} = D_{remote}$ in MB	1	1	1
BW_{LAN} in Mbps	100	100	100
rep	0.5	0.5	0.5
np	0.5	0.5	0.5
$sl(t)$	0.2	0.5	0.9
f_{LAN} = (1-$sl(t)$) * (1-np)	0.4	0.15	0.05
$dc(D)$	-	0	0
BW_{WAN} in Mbps	-	10	10
$nl(t)$	-	0.7	0.75
f_{WAN} = (1-$nl(t)$) * (1-np)	-	0.15	0.125
$Time_{dist}$ in s	0	0.1	30 (tape mount etc.)

Table 4.2: All the static and dynamic parameters which are required for equation 4.1 are listed here. The parameters for WAN take into account that RC1 disk is the local site and CERN disk1 and CERN tape are remote sites.

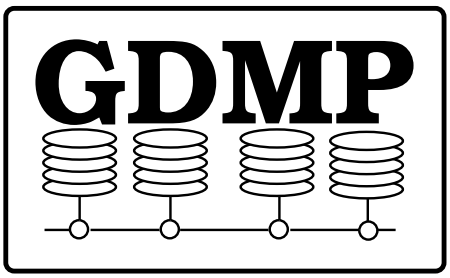
for tape access. In the replica selection cost functions in Chapter 8 we go into further details and thus a slightly more complex cost model is presented which also takes into account that accessing an object remotely and than copying it to the local site can be less time consuming than accessing a local object on a heavy loaded data server. Note that Chapter 8 deals with files whereas the model here is for single objects. In case of files, it is more important to include the load of a file server, especially if it is a dedicated FTP server that serves files to a large community in the Data Grid.

4.3.7 Concluding Remarks

The OLT proposed here is different to a classical replica catalogue that only maps logical to physical names. Our proposed OLT uses a built-in optimiser and makes use of the order of objects for determining which replicas are optimal to access first. Thus, it combines replica catalogue and replica selection. Scalability is gained by limiting the lookup operations. Finally, the OLT also supports scalable update operations by making use of the Collection Contents index (CCI) [60].

Chapter 5

Grid Data Mirroring Package (GDMP)



The GDMP client-server software system is a generic file replication tool that replicates files securely and efficiently from one site to another in a Data Grid [6] environment using Globus Grid tools. In addition, it manages replica catalogue entries for file replicas and thus maintains a consistent view on names and locations of replicated files. Files to be transferred can be of any particular file format and GDMP treats them all in the same way. However, for Objectivity database files a particular plug-in exists. All files are assumed to be read-only.

5.1 Introduction

The GDMP [108, 52, 120] project (originally called Grid Data Management Pilot) was started in early 2000 as a pilot project to evaluate the Globus toolkit, take useful features for a file replication system and produce a prototype to be evaluated in a real production environment. In early versions (until release 1.2.2 of the software), GDMP was restricted to replicate Objectivity files but in the new release 2.0 any kind of file format is accepted.

The driving force for the research and development on GDMP was twofold: on the one hand, at CERN first efforts for the European DataGrid project were under way and first state-of-the-art studies in Grid computing needed to be done. On the other hand, the CMS experiment required data transfer tools for the distributed production of simulated data, in particular for the High Level Trigger studies. Once the DataGrid project was kicked off in January 2001, the experience gained from GDMP was used and the current version of the software system is part of the DataGrid software and in use in the first DataGrid testbed. Currently, the software development process is well advanced and the project is now a collaboration between the European DataGrid (in particular the Data Management work package) and the Particle

Physics Data Grid (PPDG) [95].

With respect to the data management architecture in Section 4.1, GDMP is a replica manager with additional high-level, application specific functionality like integration of Objectivity database files to an Objectivity federation, a notification system for remote sites and an interface to mass storage systems. In this chapter we first describe the requirements for a file replication system, give details on the architecture including all Grid components that are used from the Globus toolkit and then explain interfaces and tools of the current GDMP release version 2.0.

5.2 Requirements and Basic Functionality

Although the CMS detector is scheduled to take data in year 2006, simulated production is done already now in order to test and prepare physics software and storage systems for the upcoming data challenge of several Petabytes of data a year. As regards the High Level Trigger study in CMS, simulated data is created in several regional centres like Fermilab (Chicago, USA), Pisa (Italy), Moscow (Russia), Lyon (France), Bristol (UK) in the CMS Data Grid and then needs to be transferred to other sites in the Grid. Data is either stored in Objectivity databases or in special, HEP specific file formats.

Although Objectivity provides a data replication option, it is not optimised for wide-area replication and thus a replication system was required that replicates Objectivity files as well as files of any other file format.

5.2.1 Distributed Data Production in CMS

The preparations of distributed data production have been going on for a couple of years. A data transfer production system based on Perl scripts, HTTP and secure shell copy transfers has been put in place [131]. The purpose of these scripts is to allow a user to see the contents of an Objectivity's federated database catalogue from anywhere with WWW access, without the need to run any part of the Objectivity software locally. Furthermore, the script software allows for data transfer of not only Objectivity but any kind of file, for instance Zebra-fz or Root [106] files. These Perl scripts have been the initial input for the architecture and design of GDMP. The idea was to extend the functionality by introducing a security environment and providing a fully automated replication mechanism based on a Grid infrastructure using Globus. Originally, GDMP was a Grid-enabled successor of these Perl scripts but then has been extended with several replication features, preliminary mass storage management and meta-data management for replicated files.

Since October 2000, the first versions of GDMP are in use in the CMS production environment. We have proven that Data Grid tools can be used efficiently in a production environment and thus have made a pioneer step towards the vision of a Data Grid. What is more, GDMP is now a generic file replication tool and available in the DataGrid software to serve the HEP, Bio Informatics and Earth Observation communities.

5.2.2 A Basic Usage Scenario

GDMP's new name[1] derived from the fact that the original status of a pilot project has been extended and now better identifies the functionality of the replication service.

The core functionality of the software system is to (partly) mirror one data repository to remote sites in a Data Grid. When new files are added at one site, tools have to detect which files are new in the repository and notify other sites about the availability and location of a set

[1] Grid Data Mirroring Package instead of Grid Data Management Pilot

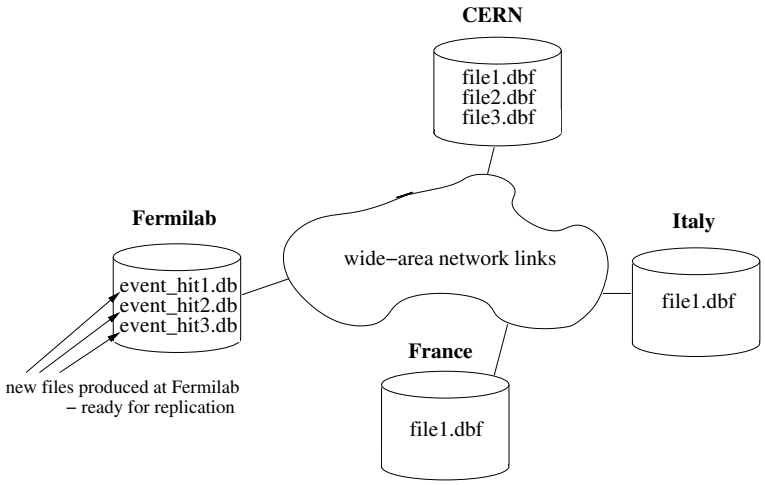


Figure 5.1: A sample Data Grid with 4 sites where new data is produced and replicated.

of new files. Figure 5.1 depicts a simple replication (mirroring) scenario in a Data Grid with four sites.

For instance, at Fermilab a set of three new files is created and should be made available to remote sites. A user at CERN shall now have an automatic tool that finds out about newly created files, transfers them to a specific storage location and then possibly updates local file catalogues (as it is the case for Objectivity files). Since Objectivity provides an internal catalogue for all local files, newly created files can be easily identified. For files of other data formats, a site catalogue is required which holds all files that are stored within a site and should be published to remote sites in the Grid. At each site, files normally reside on disks but may be partially available also on tapes and accessible via mass storage systems.

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 that has a catalogue of database files internally. 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 catalogue. 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 catalogue information. Simply put, this is done by a GDMP server running at each site where files are produced and possibly replicated.

5.2.3 The File Replication Process

The entire file replication process does not only contain a simple file copying from one site to another but contains several steps and requires to keep track of filenames and locations. The latter is achieved through a replica catalogue service. Based on the file type or format (Objectivity file, Oracle file, plain text file, etc.) a successful file replication process consists of the following steps:

- *pre-processing*: This step is specific to file formats 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

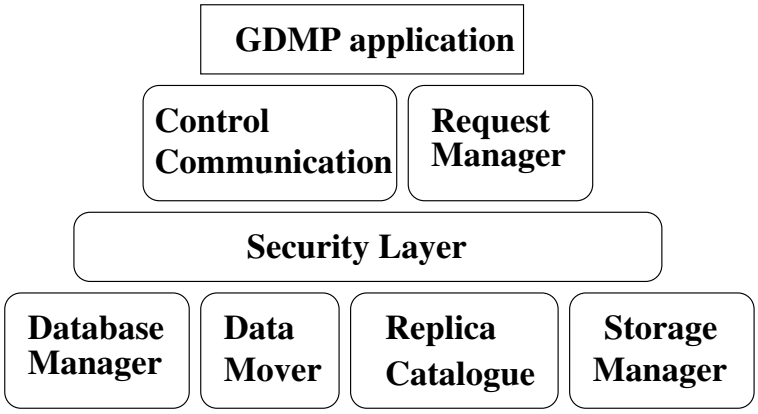


Figure 5.2: The layered GDMP architecture.

into the existing Objectivity federation.

- *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 to an internal file catalogue.
- *insert the file entry into a replica catalogue*: 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.

The GDMP replication process is based on the producer-consumer model: each data production site (producer) 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.

5.3 Architecture

We now give details on the architecture, the components used by GDMP and some design considerations. In principle, the GDMP software consists of several modules that in turn can use established Grid services like a replica catalogue service or the file copier as outlined in Section 4.1.

GDMP has a layered architecture model (see Figure 5.2) and uses several Globus libraries. The boxes represent single modules that are explained in more detail in the following subsections. Each of the GDMP client applications (command lines tools for publishing site catalogues, replicating files, etc.) as well as the GDMP server use several of the modules.

5.3.1 Control Communication

This module takes care of the control communication between the clients and the servers, and uses the Globus IO library as the middle-ware and builds high level functionality on top. It takes care of the intricacies related to socket communication over the wide area between nodes with heterogeneous architectures. The functionality includes starting and stopping the server,

connecting and disconnecting the client to and from the server and sending and receiving messages at both the client and server ends. This module provides services to other modules.

Globus IO is a thin layer on top of basic socket communication and provides APIs for easily managing sockets over TCP or UDP. The Globus Data Conversion library is used for providing high level interfaces that are similar to RPC calls.

5.3.2 Request Manager

Every client-server system has to have a way to generate requests on the client side and interpret these requests on the server side. The Request Manager module does exactly that. It contains several request generator/handler pairs and more can be added for customised use. This module is based on the Globus Data Conversion library which provides methods to convert data between different formats supported by variable machine architectures. The requests are generated on the client side by filling a buffer with the appropriate function handler, which is to be called on the server end, and any arguments required by that function. On the server side this buffer is unfolded and the data types are converted according to the local architecture. Finally, the respective function is called with the given arguments. The Request Manager Module basically mimics a limited Remote Procedure Call (RPC) functionality with the advantage of being light-weight and extensible.

5.3.3 Security Layer

Security is one of the main concerns in a Grid. Allowing people from outside one's domain to use the resources is a big issue for most organisations. Sensitivity of data and unauthorised use of network bandwidth to transfer huge files are the main security issues we are dealing with. This is done by the security layer. It is based on the Grid Security Infrastructure (GSI) [42] which is an implementation of the Generic Security Service (GSS) API. It uses the Public Key Infrastructure as the underlying mechanism. The security module provides methods to acquire credentials, initiate context establishment on the client side and accept context establishment requests on the server side (context is established to authenticate the client), encrypting and decrypting messages and client authorisation. The server authenticates and authorises any client before processing its request, hence, the software protects a site from any unwanted file transfers and blocks any unauthorised requests. Consequently, every client server communication is done via secure communication. This also applies for the file transfer itself. However, data encryption for file transfer is not used.

5.3.4 Database Manager

This module interacts with the actual database management system if database files are replicated. The module relies on the APIs provided by the particular data storage system being used. In our case the DBMS of choice is Objectivity/DB, hence we use Objectivity's command line tools to implement this module. If a new data type, i.e. a new database management system, is used, this is the only module which has to be swapped by the one which can interact with the specific DBMS. The functionality includes

- create Objectivity federation (including database schema)
- schema upgrade based on schema of a remote Objectivity federation
- retrieve the database catalogue (or file catalogue), containing information about the files currently present in the database,
- attach files to the DBMS once they arrive on the client side where they are validated.

5.3.5 Data Mover

The main purpose of GDMP is to replicate files over a wide-area network in an automatic, efficient and fault tolerant way. This module is responsible for the actual transfer of files from one location to another.

The Data Mover uses the GridFTP client library of Globus for file transfer and relies on GridFTP servers (based on the WU-FTP server with Globus modifications for GridFTP). Since GridFTP uses the Grid Security Infrastructure (GSI), we have the same security mechanism for both the Control Communication and the Data Mover. The functionality includes client authentication and authorisation before the transfer starts, transferring files to and from the server, validating the transferred files using the file size and checksum information, resuming the file transfer from the latest checkpoint after a network failure[2], using a progress meter to output the progress during a file transfer and finally logging all file transfers that have been completed.

Point-to-point replication is on of the major performance criteria for a large Data Grid. We require high-performance data transfer tools which is the target of the Globus Data Grid toolkit's GridFTP [4] system. GridFTP allows for parallel and striped transfer streams, TCP tuning and is based on GSI. In [120] we present the results of detailed performance studies conducted with the an early alpha GridFTP release.

The GDMP Data Mover service has a layered, modular architecture so that its high-level functions are implemented via calls to lower-level services that perform the actual data manipulation operations. In this case, the lower-level services in question are the data transfer services available at each site for movement of data to other Grid sites.

5.3.6 Replica Catalogue

GDMP uses a replica catalogue service to maintain a global file name space of replicas. GDMP provides a high-level replica catalogue interface and currently uses the Globus replica catalogue as the underlying implementation. An end-user who produces new files uses GDMP to publish information into the replica catalogue. This information includes the logical file names, logical file attributes (such as file size and modify time-stamps) and the physical location of the file. In detail, when a site publishes its files:

- These files (and the corresponding file attributes) are added to the replica catalogue.
- The subscribers are notified of the existence of new files (more details on the subscription model are given below).

The replica catalogue module also ensures a global name space by making sure that all logical file names are unique in the catalogue. GDMP supports both the automatic generation and user selection of new logical file names. User-selected logical file names are verified to be unique before adding them to the replica catalogue. Race conditions on the replica catalogue are currently not dealt with.

Client sites interested in a new file can query the replica catalogue to obtain the information required to replicate the file. Users can specify filters to obtain the exact information that they require; information is returned only about those logical files that satisfy the filter criteria. The information returned contains the meta-information about the logical file and all the physical instances of the logical file. This information can then be used as a basis for replica selection based on cost functions and is discussed in detail in Chapter 8.

[2]In an early version of GDMP we have used a different GSI enabled FTP client library that provided this feature. It is also available in GridFTP and we plan to use it as soon as it is a public release.

The current Globus replica catalogue implementation uses the LDAP protocol to interface with the database backend. We do not currently distribute or replicate the replica catalogue but instead, for simplicity, use a central replica catalogue and a single LDAP server for the replica catalogue service. In the future, we will explore a more scalable version of the replica catalogue and presented first results in [121].

The GDMP replica catalogue module is a higher-level object-oriented wrapper to the underlying Globus replica catalogue library. This wrapper hides some Globus API details and also introduces additional functionality such as search filters, sanity checks on input parameters, and automatic creation of required entries if they do not already exist. The high-level API is also easier to use and requires fewer method calls to add, delete, or search files in the catalogue.

5.3.7 Storage Manager

In order to interface to Mass Storage Systems (MSS), the GDMP server uses external tools for staging files. For each type of Mass Storage System, tools for staging files to and from a local disk pool have to be provided externally. We assume that each site has a disk pool that can be regarded as a data transfer cache for the Grid and that, in addition, a Mass Storage System is available at the same site but does not manage the local disk pool directly. The staging to a local cache is necessary because the MSS is mostly shared with other administrative domains, which makes it difficult to manage the MSS's internal cache with any efficiency. Thus, GDMP needs to trigger file staging requests explicitly. This is our current environment, which might change slightly in the future.

A file staging facility is necessary if disk space is limited and many users request files concurrently. If a remote site requests a replica from another remote site where the file is not available in the disk pool, GDMP initialises the staging process from tape to disk. The GDMP server then informs the remote site when the file is present locally on disk and at that time performs automatically the disk-to-disk file transfer.

In the replica catalogue, physical file locations are stored and contain file locations on disk. Thus, by default a file is first looked for on its disk location and if it is not there, it is assumed to be available in the Mass Storage System. Consequently, a file stage request is issued and the MSS transfers the file to the disk location stored in the replica catalogue. Note that Objectivity has an interface to HPSS [54] and the file naming convention is the same: the default location is a disk location. Some other storage management systems have a tape location as a default file location.

Note that more sophisticated space management mechanisms such as reservation of disk space are currently not available but are easy to add [40]. In particular, the underlying storage system needs to provide an API for storage allocation, e.g., `allocate_storage(datasize)`. In this case, the file replication transfer might be started only if the requested storage space can be allocated.

5.4 The Data Replication Process and Policies

We present the data replication process and also introduce some GDMP client applications which are then described in more detail in the next section.

5.4.1 File Catalogues

GDMP uses a few catalogues that are used for internal book keeping and monitoring of the replication process. We elaborate on how GDMP manages the file transfer and illustrate the

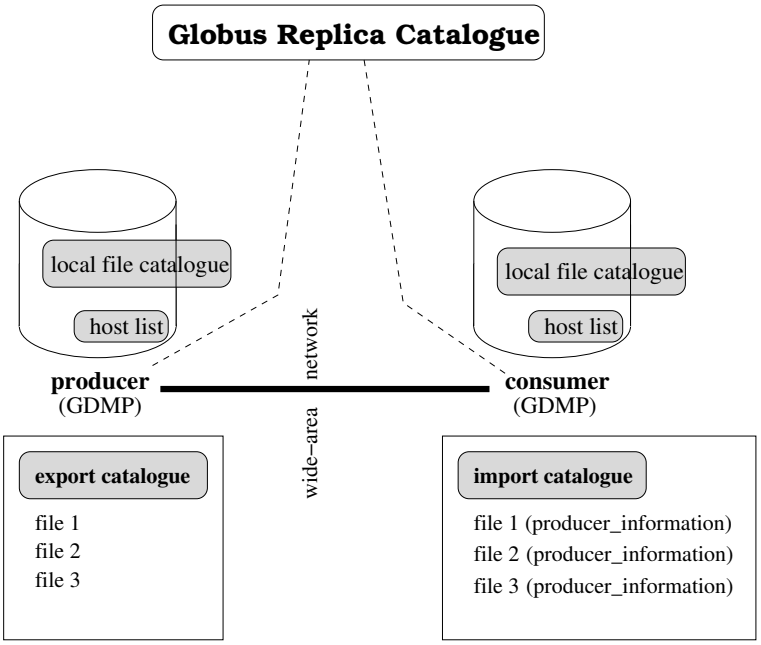


Figure 5.3: The role of the local file, export, import catalogues

data flow by a producer-consumer example. The producer is the site where one or several files are written, and the consumer is the site that wants to replicate (receive) these files locally. Once the producer has finished writing a set of files (or just a single file), every single file needs to be registered in a *local file catalogue* which only contains files that are available at the local Storage Element. The catalogue contains the physical filename and logical file attributes like logical filename, file size, creation time, CRC checksum, file type. The local file catalogue is hidden from outside users and is thus only visible to the local GDMP server. The client application `gdmp_register_local_file` is used for inserting files to this catalogue.

At a certain point in time, the producer can decide to publish its local files to other Grid sites using `gdmp_publish_catalogue`. In detail, all file entries of the local file catalogue are written into the replica catalogue[3] and also sent to subscribed consumers at remote sites. A list of all newly published files and their related information is written to a local *export catalogue*. The consumer creates an *import catalogue* where it lists all the files that are published by the producer and have not yet been transfered to the consumer site. The import catalogue holds the host name of the FTP server and all related physical and logical file information for each file. Figure 5.3 illustrates this model graphically and shows that a GDMP installation (including client and server applications) is required on each Storage Element taking part in the data replication process.

When files are published, all required file information is read from the local file catalogue. Since the catalogue holds file attributes like size, during the execution of the publish command the files to be published do not need to reside on their physical location on disk but can already have been staged to a mass storage system. Note that files have to be at the disk location when `gdmp_register_local_file` is called since file size and CRC check sum are automatically created by GDMP and then stored in the local file catalogue.

[3] An insertion to the replica catalogue can also be disabled.

5.4.2 Subscription and Replication Model

Although replica information is available in the replica catalogue, one GDMP site can subscribe to a remote site in order to get notified when new files are created. Thus, a site keeps a *host_list* catalogue (see Figure 5.3) where all hosts are listed that are subscribed to newly created local files.

The currently implemented replication policy is an asynchronous replication mechanism with a subscription model. A producer can choose at what time new files are written into the export catalogue and thus made publicly available for consumers. Hence, the producer can decide the degree of data consistency by delaying the publication of new files. In the example above we only have one consumer for demonstration purpose. In reality, the number of consumers can be large and depends on the number of sites in the Grid. The subscription model enables that the subscribed consumers get informed immediately when new files are published in the export catalogue. Each consumer that wants be be notified about changes in the producer's export catalogue, subscribes to a producer. Depending on the degree of interest in a producer's data, consumers might want to subscribe to only some of the producers in the Grid.

Since the data exchange has to be done in a controlled and secure way, a consumer first has to be a "registered Grid user" at the producer site, i.e. the user has to be added to the grid-mapfiles[4] of the producer sites. These files contain all the users who are allowed to talk to GDMP servers running on a site. Thus, a producer has total control over who subscribes to and transfers files from its site. Once this is done, a consumer is allowed to subscribe to the producer site. The producer then adds the new consumer and its related information in the host_list catalogue.

The consumer can decide when to start the file transfer from the producer to the consumer site with the tool `gdmp_replicate_get`. The tool reads the import catalogue and starts FTP sessions and interactions with remote GDMP servers to get the necessary file.

In our example above we distinguish between a producer and a consumer site in order to illustrate the data flow. However, each site in a Data Grid can produce new files and get files from other sites, i.e. a site can be producer and a consumer. Thus, each site manages an import catalogue where it stores available file information from other sites, and an export catalogue where it publishes files created by itself.

5.4.3 Notification System

GDMP provides a powerful notification mechanism which can be used to monitor the replication process as well as the publishing of new information. Configurable scripts are used for interfacing user specific applications.

When a user publishes a local file catalogue with `gdmp_publish_catalogue`, a remote server gets notified and calls a configurable script which can then be used by external programs to start a data transfer request. In principle, `gdmp_replicate_get` can be executed and a fully automatic replication process can be set up. On the other hand, a similar notification script is called at the producer site when a consumer has successfully replicated a file. Thus, producers can keep track of consumers requesting and replicating files and can delete files again if local storage space is required.

5.4.4 System States for File Replication Process

In the entire replication process, we distinguish several system states that are indicated by files with particular extensions. Note that these files are only kept for local book keeping and are

[4] Globus specific file which is used for user authentication

not accessible to remote users. The filename of "file" corresponds to a logical filename of the actually transferred file and can be retrieved from the import catalogue. Thus, one can check the current status of a file transfer and discover possible problems.

The following transfer states are defined and is indicated by a file with a particular extension:

- file.**stat**: the transfer of a file is currently going on or the file has been requested for transfer.
- file.**replicated**: a file has been copied to its destination but not yet validated nor registered.
- file.**validated**: a file has been validated locally after the copy process. The validation contains a file size check and CRC checksum a comparison with the original file at the remote site.
- file.**registered**: a file has been successfully validated and registered to the local file catalogue.
- file.**transferred**: the file has already been successfully transferred, validated and registered but not yet deleted from the import catalogue. The tool `gdmp_catalogue_cleanup` needs to be used to delete these files plus the entries in the import catalogue. Once the file is in "transferred state", it can be used by applications.
- file.**notified**: a remote site has been notified about the correct file transfer (including validation and registration).
- file.**req**: a file is requested from a remote site and is on tape rather than on disk. A staging request is initiated.
- file.**done**: This file only exists for a very short time: when a staging requests has been done at a remote site (from MSS to disk) and the local server can start to get a file. Once the file transfer starts, the file is renamed to **.stat**.

In addition to the successful states, we also have a few error states:

- file.**not_validated**: file validation failed
- file.**not_registered**: file is not registered to the local file catalogue
- file.**not_notified**: file notification failed
- file.**error**: an error has occurred during the attachment of Objectivity files.

5.4.5 Partial Replication: Filtering Files

GDMP allows a partial-replication model where not all the available files in a federation are replicated. In other words, one or several filter criteria can be applied to the import and/or export catalogue in order to sieve out certain files. For instance, a site only wants to make files publicly available that contain the word "Muon" in the filename. Hence, the export catalogue, which contains all the files that a site wants to publish, has to be filtered and files that do not contain the filter criterion are deleted. A site that wants to get files from other sites can as well choose which files it wants to get. In this case the filter has to be applied on the import catalogue. This allows for a partial replication model where the producer as well as the consumer can limit the amount of files to be replicated.

5.4.6 Fault Tolerance and Failure Recovery

In a distributed system we can identify several sources for errors. On the one hand, there are hardware errors like a physically broken network cable, a broken network card, processor or any other piece of hardware. On the other hand, a part of the software system can have a failure. For instance, the FTP server dies, the file system crashes or some other part of the GDMP software does not work properly. In any of these cases the connection between the distributed clients and servers is broken. We refer to such a failure as the *connection is broken* and do not distinguish between the different reasons for this failure. We emphasise that the communication is broken and the message or control flow cannot be continued. For instance, in case of a broken connection anywhere between site A and site B, site A cannot publish its catalogue to site B.

In the current version of GDMP, each site is itself responsible for getting the latest information from any other site. This is also expressed by the subscription system, where a site has to explicitly subscribe to another site in order to get the file catalogue. Furthermore, when a site recognises a local error which has caused the broken connection, this site has to request the required information from the peer site. A site which publishes information to a subscribed site does not re-send information nor logs that a site could not receive the information. A site can retry to send the message again to the destination site within a particular time frame which can be set by a timeout parameter. If the re-sending fails again, the sending site stops trying to contact the sites and hands over the responsibility to the destination site to recover from the broken connection.

To sum up, the policy is the following. Each site has to be aware of its state (connection okay or broken). Then it has to search for the origin of the broken connection. If it detects that the error is on the local site, it has to recover otherwise the peer site is responsible for failure recovery.

A site can be unavailable for several hours or even days. Meanwhile several producers can have created and published files, and the entries in the export catalogues may already have been overwritten. However, published replica information is also available in the replica catalogue. A consumer can recover from the site failure by issuing the command `gdmp_get_catalogue`. Once a producer site has published its catalogue, the catalogue is available to be transferred to any subscribed consumer. GDMP at the consumer site then compares the consumer and producer catalogue and creates the necessary information in the consumer's import catalogue. Possible multiple appearances of files are deleted in order to keep the import catalogue's entries unique.

Only the file which is currently been sent when the network connection breaks has to be resent. Since the FTP client has a "resume transfer" feature, not even the whole file has to be transfered but only the part of the file that is still missing since the last check point in the file. This allows for an optimal utilisation of the bandwidth in case of network errors.

Other possible errors: A site may publish a file several times and export it to other sites. In order to have files only uniquely transferred, each site checks automatically if the file in the local import catalogue does not already appear in the federation catalogue. Furthermore, several sites can publish the same file to a specific site two times. Consequently, on creation of the import catalogue, the system checks if the file to be entered is unique. Only if this is the case, a new file is inserted into the import catalogue.

5.5 GDMP Applications and Interfaces

In this section we go into more details on concrete GDMP tools and their usage. The client-server software consists of a GDMP server that needs to be installed and running on each site

taking part in the data replication process and several client applications.

5.5.1 GDMP Server

The GDMP server needs to run constantly and serve several user requests from GDMP client applications. We do not expect a very high load on the server (less than three requests per second) and thus do not require a high performance server, but it needs to be robust.

Currently, a GDMP server is configured to run via the Internet daemon (inetd) and can be started on any user defined port. Simply put, the inetd provides Internet service management for a network. It listens on certain ports and calls other servers or daemons for serving the request. As regards GDMP, we register a certain port, e.g. port 2000, with the Internet daemon and when a GDMP client connects to the machine via a socket connection, the Internet daemon takes the request on port 2000, starts the GDMP server and passes all the socket information to the GDMP server. The GDMP server in turn handles the client request. By default, GDMP servers are supposed to run on port 2000 but can be configured to run on any other port that is not hidden by a firewall.

All interactions between GDMP clients, the GDMP server and the underlying FTP server are done via GSI. In detail, every client request that comes over a socket first needs to be authorised and authenticated (the local grid-mapfile is used for that) and then the Request Manager module serves the client request. Like any client application using Globus, also a server requires a security proxy to be running. The server uses a dedicated server certificate which is included in the GDMP software distribution. Thus, when a server is started, the required Grid proxy is gained automatically. Any output or error message from the server is logged in a corresponding log file.

5.5.2 GDMP Client Applications

GDMP offers a set of client applications to register new files to a local catalogue, publish a set of newly created files to subscribed hosts, start the actual file replication process, and subscribe to a remote GDMP server. In addition, the software package contains a few other administration tools. We describe briefly the basic functionality and refer to the GDMP User Guide [122] for further details. We illustrate the sequence of client operations that is necessary for the replication process:

1. All GDMP client applications need to use GSI and thus require a X.509 certificate issued by a trusted certificate authority (either Globus, DataGrid or any national authority). Before any client application can be started, a client has to have a Globus proxy running[5] in order to start the authentication process handled by the GDMP security module internally. We use Globus' *single login* procedure: once a client has successfully got the proxy on one machine, requests can be sent to any GDMP server (given that the X.509 distinguished name is included in the remote server's grid-mapfile) without any further password to be entered.

2. Once the proxy is gained, the GDMP sites need to subscribe to each other. With a subscription, one site expresses its interest to get notified about newly created files. The following client application `gdmp_host_subscribe` has to be used providing remote host name as well as the remote port name as parameters. In order to be sure that a remote server is running and listening on the correct port, the tool `gdmp_ping` can be used to probe the remote server.

[5] A proxy can be gained via the Globus tool `grid-proxy-init`.

3. In the next step, files need to be registered in the local file catalogue using the command `gdmp_register_local_file` which registers either individual files or entire directories.

4. Once files are registered locally, they can be published to all subscribed sites with the tool `gdmp_publish_catalogue`. Note that this does not include any file transfer but only replica and location information are exchanged.

5. At the remote site, a server receives a notification message about all new files and can also further call a script locally. In principle, once a remote server is notified, the file transfer process can be started with `gdmp_replicate_get` and all files from the import catalogue are requested from remote sites and transferred to the local site. All transfer information like file size and network utilisation is logged in a log file. Several parallel clients can be started too in order to optimise network traffic for un-tuned network connections[6].

For all the client commands filters can be applied and one has to distinguish between flat files and Objectivity database files. Filters on import and export catalogues can be assigned with the tool `gdmp_filter_catalogue`.

5.6 Performance Considerations and Results

There are several ways to evaluate a file replication system and we just mention a few of them: does it satisfy the user requirements? Is it secure enough and can one trust users at distributed sites in the Data Grid? Is the performance okay? In the development process of GDMP we first put emphasis on functionality. Since Grid tools are still rather new, this question is reasonable and we showed that Grid tools can be used for file replication and at the same time satisfy basic user needs. Once the file transfer works correctly, one can start to optimise the actual transfer of data over the network.

GDMP is designed to replicate a set of files to a remote site. In principle, there are two basic methods to optimise file transfer for several files in a multi-user environment with data traffic over the Internet:

- optimise the transfer of a single file and get as much network bandwidth as possible: This is achieved by using several parallel streams for a single file transfer and optimised TCP buffer sizes. We report on details of TCP buffer size tuning in [120]. It has the advantage that a single file gets transferred fast by using a large percentage of the possible throughput of a network link but has the disadvantage that other, concurrent network transfers get lower bandwidth rates.

- parallel client transfers: This is the conventional and easier way to transfer a set of files efficiently. Our experience has shown that 5 to 10 parallel client transfers (using gdmp_replicate_get) can also saturate the throughput of a network link. As our results indicate, this is true for wide-area connections but not really significant for local-area transfers.

Here, we only report on performance results of the parallel clients and refer the to [120] and [4] for GridFTP performance results for parallel streams within a single client transfer.

GDMP 2.0 has been developed and tested on Linux RedHat 6.1 and 6.2 using the C++ compilers gcc-2.91.66 and gcc-2.95.2. We measured the latency for the whole replication process of a number of files on local as well as the wide-area networks. Note that the file replication

[6] An un-tuned network connection is a data transfer link where default or non-optimial buffer sizes are used.

process consists of several steps like CRC check sum calculation, registration of files in catalogues and notification of remote sites and thus has a small overhead compared to the raw transfer of the file. Our performance results here focus on the entire replication process.

For the LAN tests within CERN's computer centre we used two Linux PCs (RedHat 6.2, kernel 2.2.19-6.2.7), 800MHz, 100Mbps network). We transferred files with the size of 273 MB and compared the transfer speed for a single client and for two parallel clients. We obtain the following result. Note that all the tests are done with default configurations of the GridFTP servers and buffer sizes on all machines.

	single file transfer	*2 parallel clients*
transfer speed in s	[91 - 134]	[220 - 240]
transfer in MB/s	[2 - 3]	[2.2 - 2.4]
validation time in s	50	150
registr./notific. in s	1	1

During the tests the machine was in use by a few users who used parts of the CPU time temporarily. This is a valid assumption also for a Grid with several users. We did several tests and the transfer results are in the span given in brackets. We can derive that for fast network connections (over the LAN) there is hardly any performance gain for parallel clients since a single client utilises already much of the network link as well as of the disk I/O rate. The maximum theoretical throughput of the disk on both machines is about 4.4 MB/s. Furthermore, the GDMP client uses a rather high amount of CPU (between 10 to 40 percent) as compared to WAN tests (see below) with about 2 to 4 percent CPU time.

The WAN test was done with one of the above CERN machines (Geneva, Switzerland) and a Linux PC (Redhat 6.1, kernel 2.2.16-3, Pentium III, 863 MHz, 512MB RAM, 100 Mbps Ethernet network card) in Fermilab, near Chicago, USA. The network link to Chicago has a theoretical maximum throughput of 45Mbps. We tested several single and parallel clients in order to get maximum throughput from the network link for a set of files. Figure 5.4 shows the pure FTP rates as well as the aggregated GDMP transfer rates including validation, registration and notification of files. The graph shows that with increasing number of parallel clients the throughput of a single client decreases whereas the aggregated throughput for several parallel clients approaches to a maximum bandwidth utilisation with about 8 to 10 parallel clients. With a higher number of parallel clients, the gained throughput is only marginal.

Our results show with checksum caching and smart dummy attach methods (a common trick to attach files to an Objectivity federation), we get almost the same aggregate throughput with GDMP as with plain, single GSI-FTP transfer stream. Hence, GDMP provides much more automated and reliable replication with tolerable overhead.

To conclude, GDMP's data mover module and thus the overall transfer performance of GDMP client transfers depends on the underlying file transfer implementation, the network topology and the tuning of the network. Thus, performance tuning is not a real issue for GDMP and thus we claim that network performance needs to be provided by underling Grid tools (e.g. a file transfer service). GDMP's responsibility is to interface with a given service.

5.7 Conclusion and Future Work

GDMP is now part of the official DataGrid software system and will be used in first DataGrid testbed in the last quarter of 2001. Furthermore, GDMP has been in use at several sites (CERN, Caltech (Pasadena, CA), Fermilab (Chicago, IL), Pisa, Moscow, San Diego, Wisconsin) in the CMS experiment to transfer almost a Terabyte of data in total. The software package is in

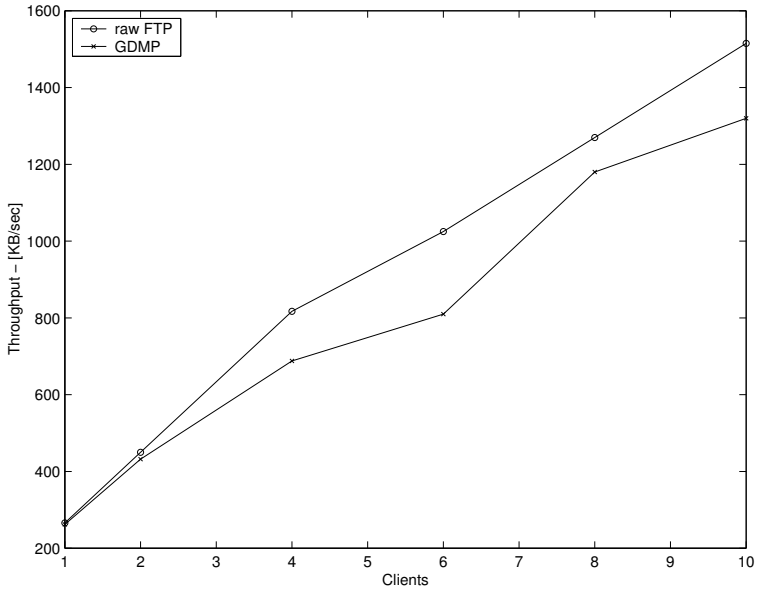


Figure 5.4: Comparison of several parallel clients transferring 200MB files over the transatlantic link Chicago-Geneva. Note that the 10 parallel clients transferred data in the morning (European time) while all other transfers were in the evening (European time). This shows again the fluctuation of network throughput. The two graphs correspond to the raw FTP transfer time (FTP) and the aggregated replication process (GDMP) including validation, registration, etc. A sinlge GSI-FTP stream gave between 220 tp 260 KBs depending on the day time.

use in DataGrid's first testbed. The GDMP is continuing and we will add more fault tolerance and failure recovery will be added.

Our experience has shown that new technology like Grid tools can be combined and used efficiently for data replication in a wide-area distributed environment. The work on GDMP is continuing and we will add more fault tolerance and failure recovery to our future releases.

Chapter 6

Distributed Replica Catalogues

In previous chapters we introduced the concept of a replica catalogue but we did not pay attention to the location of the replica catalogue. In this chapter, we point out some problems of a central replica catalogue and propose a distributed one. We deal with administratively scalable and efficient replica catalogue lookup issues in a Data Grid. However, our approach can be applied to any data intensive, distributed catalogue system [121].

6.1 Introduction

A replica catalogue maps a logical filename to a physical filename and thus is capable of resolving physical file locations. Furthermore, the replica catalogue manages the namespace of replicated files and guarantees the uniqueness of filenames. The Globus replica catalogue [4] is one existing example of a replica catalogue in a Data Grid. In the current implementation, this catalogue can be regarded as a Grid middleware service that stores logical filenames and physical file information. The catalogue itself can either be centrally available or distributed/replicated to several sites in the Grid. Since the catalogue is implemented in LDAP (Light Weighted Directory Access Protocol [136]), LDAP referrals or LDAP replication can be used for catalogue distribution and/or replication.

We propose an administratively scalable and flexible catalogue system where each site in a Data Grid manages a local file catalogue and a global replica catalogue is used for looking up sites rather than entire physical file locations. Each site has a local catalogue server for looking up exact file locations within a site. A catalogue server is an HTTP server and is identified by a URL and a port number. In particular, a client application sends a file request in form of a logical filename to a replica catalogue which returns one or more URLs of catalogue servers. An additional lookup step is required for resolving the exact physical file location. Consequently, *administratively scalable* means that as the number of catalogue entries and operations increase, the effort required to administer a Grid-based catalogue is controlled and grows no faster than the effort required to administer an isolated catalogue.

We present the design and implementation of the redirection server and the client protocol interaction. Two interfaces to a flat file catalogue as well as an Oracle 8i file catalogue are available. Special purpose re-synchronisation servers make sure that the local catalogues and the global replica catalogue are consistent. This also allows for relaxing and maintaining consistency of replica and file catalogues in the Grid.

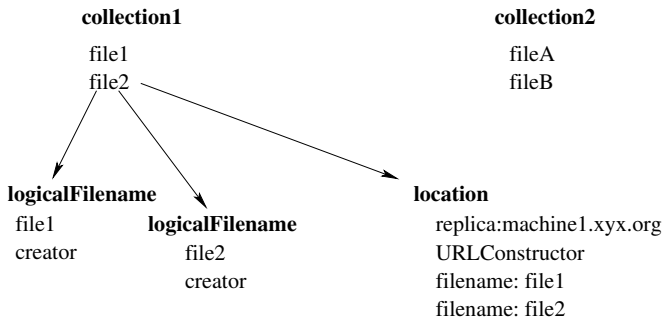


Figure 6.1: Globus replica catalogue

6.2 Replica Catalogues and Related Work

A Data Grid can be regarded as a hybrid approach between distributed file systems, Internet technologies and distributed database management systems. Related work can be found in each of these areas but none of them discusses the HTTP redirection approach of replica catalogue lookups. Our main focus is the Globus replica catalogue in Section 6.2.2.

6.2.1 Related Work in General

Lookup operations in file systems are of major interest in [78, 46]. In particular, [78] discusses lookup servers that use an Active Disk approach. This special purpose server looks up file locations whereas a disk server only needs to provide the file itself. Client caching is used to optimise the load of the lookup server.

Object migration and object redirection through HTTP redirection is discussed in [83]. The presented framework replicates "hot" objects and has the capability to redirect client requests for load balancing. Since most of the requested objects are rather large (e.g. multi media objects), the additional redirection step is negligible. Similar to our approach, the location of objects is derived from the HTTP header.

Another representative paper is [25] that discusses distributed catalogues. More related work can be found in commercial distributed file systems like NFS or AFS. However, in these products there is not as much site autonomy as in our approach. Cataloguing is an important issue also in SRB [9] and the proposed MCAT.

6.2.2 The Globus Replica Catalogue

In the Data Grid research community there exists currently one proposed replica catalogue implementation which is part of the Globus Grid computing middleware toolkit [4]. We will take this catalogue as our primary source for discussing a global replica catalogue. Based on this we elaborate on a distributed catalogue system and introduce our approach.

The Globus replica catalogue (RC) is based on the LDAP protocol and stores file information in an LDAP database backend. Thus, replica catalogue information is represented via an LDAP schema. Currently, the catalogue holds file entries for all logical and physical files managed in the Grid or by a particular Virtual Organisation (VO) in the Grid.

The RC schema is built as follows. Individual, logical files are grouped together into a logical collection where a collection can be distributed and replicated to several sites (see Figure 6.1). Each collection has a location object which holds the physical location of a file collection. In particular, the location object holds (among other possible arguments) a URL constructor that can be used to build the entire physical file information for a file replica. Thus, the URL

constructor is used to generate the physical file name. This allows for a mapping between a logical and a physical filename. Consequently, the RC knows about all the storage locations in the all the Grid sites. For each location a URL constructor identifies the storage location uniquely. An example of a physical filename (including the URL construction information) can look as follows. Note that the protocol name is added to the URL constructor which gives us the flexibility to add another protocol for identifying how to access a file (see Section 6.3.4):

```
gridftp://machine1.domain.org/dir1/file1
```

GridFTP [4] is an extended version of FTP and is optimised of fast file transfer. However, any other file transfer protocol can be used in the URL constructor.

6.2.3 Motivation for a Distributed Replica Catalogue

The main contribution of our work is to distribute parts of the central replica catalogue information to sites that actually store files. This gives each site the flexibility to move files locally for load balancing or backups without notifying a central replica catalogue. In an environment with lots of file movement there is less load on a central catalogue and thus the bottleneck for central update requests can be reduced.

Another motivation for our redirection approach is additional fault tolerance. A central catalogue server might fail and thus be inaccessible for lookups. Certainly, within a site the server can be replicated and the catalogue service is thus more fault tolerant. However, this is no solution if the wide-area network connection to this site is down. In our approach, catalogue information is distributed and if some sites fail, there is still an entry point to local replica catalogues and local files.

In order to make a catalogue architecture administratively scalable, the number of possible interactions during catalogue updates must be tightly bounded. This architectural philosophy has at its core the notion that the fewer update operations interact, the less need there is for administrative (i.e., human) intervention when an update operation fails. In the distributed catalogue architecture proposed in this paper, each site catalogue can be maintained and modified independently of any other catalogue. Thus, once an entry is registered in a higher level catalogue (i.e., replica catalogue), that entry need not change even when the file changes location or its attributes. Not only does this effectively eliminate catalogue traffic when site catalogues change for administrative reasons; it also drastically reduces the need for administrative intervention when an update operation fails and leaves the system in a possibly inconsistent state. In fact, because each site catalogue is essentially independent and merely federated through the replica catalogue, the amount of effort required to administer a federated site vs. a non-federated site is within a constant factor. Hence, the system is administratively scalable.

6.3 Architecture of a Distributed Catalogue Approach

We believe that an administratively scalable replica catalogue architecture can be provided by distributing the catalogue and giving certain autonomy to individual sites to maintain local catalogues. In this section we elaborate on our proposal for a powerful catalogue distribution and lookup mechanism. Note that the proposed catalogue is supposed to be used for a single Virtual Organisation rather than an entire Data Grid.

6.3.1 Catalogue Distribution

One can distribute the catalogue in a way that each site has to manage its own, site specific file catalogue and the global replica catalogue only knows about particular sites and not the

detailed storage location of the files within a site. Thus, each site has the flexibility to move files within the site without informing the replica catalogue about site specific file movements, machine names, storage locations etc. This also means that an additional mapping step is required when a client application wants to resolve the logical filename to a physical one. We call this additional step the **URL-map step**. In summary, the benefits of our approach are as follows:

- Files can be stored in local catalogues only and do not need to be made available to the Grid.
- Flexibility and site autonomy: if a site moves files to different locations within a site for load balancing reasons, updates of file locations only need to be done at the local file catalogue and not in the global replica catalogue. This reduces the load on the global replica catalogue and allows for more scalability than a single, central replica catalogue.
- The management of the distributed catalogues is easier since update propagation problems are mostly dealt within sites.

This approach requires that each site has a catalogue server (catServer) that has access to a local (site specific) catalogue where all files are listed that are stored within the site. The following file information has to be stored in the site specific catalogue:

- logical filename
- physical filename
- indication if the file is registered in the global RC

For the local, site specific file catalogue we do not require any specific implementation. The catalogue can be stored in any format (flat file, database backend, etc.). The catalogue server only needs to have access to the entries of the catalogue. In the catalogue server a simple object-oriented approach is used with a class for interfacing any catalogue that might exist. Plug-ins for particular catalogue implementations can easily be added. A file catalogue in a flat file format looks as follows:

```
dir1/file1::gridftp://flora05.slac.stanford.edu/dir1/file1::r
dir1/file2::gridftp://flora07.slac.stanford.edu/dir1/file2::r
dir1/file3::gridftp://flora12.slac.stanford.edu/data/dir1/file3::u
```

Note that the logical filenames here are stated for test reasons and do not satisfy any convention for logical filenames. For more details on logical filenames and their uniqueness refer to Section 6.4.2. In general, the client application provides a meaningful logical filename. The catalogue just needs to guarantee that filenames are unique and has to issue an error message when duplicate logical filenames are entered into the catalogue. The file triple in the catalogue is only:

```
LFN::PFN::RINFO
```

where LFN is logical filename, PFN is physical filename and RINFO is the registration information.

In order to discuss the mapping step, we illustrate this by an example (see Figure 6.2). We assume a client application wants to access replicated data via a replication tool that makes use of a replica catalogue to get a physical file. For example, the client could issue an FTP

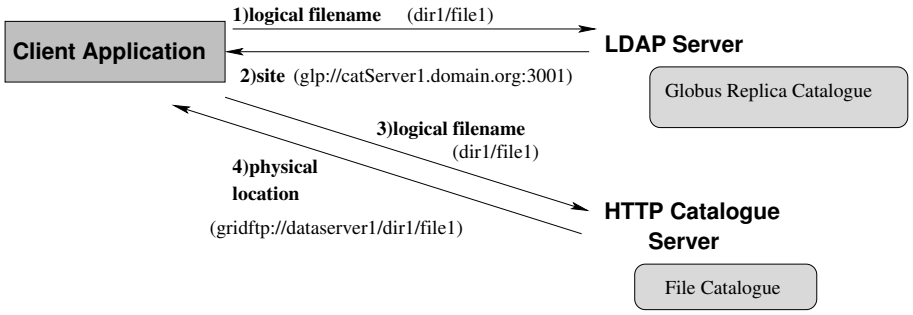


Figure 6.2: Client server interaction

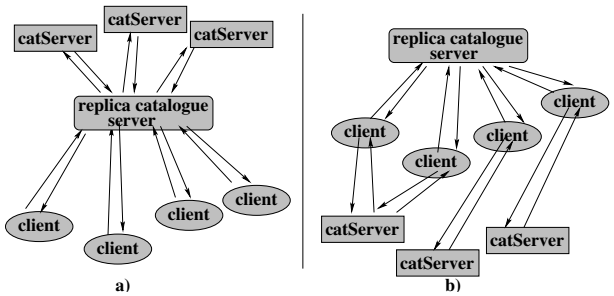


Figure 6.3: URL map step at the server side (a) and at the client side (b).

GET command. The client program first sends a logical file name to the replica catalogue in order to receive the site where the corresponding physical file is stored. Now the URL-map step is required to get the exact physical location of the particular file. This mapping step can be done directly on the machine that hosts the replica catalogue or by the client machine. Since we assume that the RC machine is a bottleneck for this mapping step if too many clients connect at the same time, the mapping step is done in the client application. This means that the client application sends a URL map request to the catalogue server at the site where the physical file is located. The catalogue server is then responsible for serving client requests for the URL-map step. The two possible interactions methods are depicted in Figure 6.3. Our approach does not remove the entire bottleneck but the load on the machine that establishes the bottleneck. Note that clever replication of the replica catalogue server can reduce more significantly the bottleneck of the replica catalogue server.

6.3.2 HTTP for Interaction with the Catalogue Server

We have chosen HTTP as a protocol for file location lookups since the protocol is well understood, widely supported, and sufficiently expressive to provide all of the functionality needed to interact with the catalogue server. Perhaps the most important aspect of the protocol is it's ability to support various forms of redirection, the underpinning of the mechanism that the catalogue server uses to direct clients to the actual location of the data. HTTP also provides a simple mechanism to inform the client how that data must be accessed via the protocol identifier that is included in the redirecting URL, naturally accommodating a heterogeneous environment. Because HTTP is widely supported, installations can build distributed catalogues with little more effort than installing a web server. These aspects of HTTP make it a natural protocol for finding distributed objects.

Consequently, the URL map-step interaction is done in HTTP whereas the client interaction

with the replica catalogue is done in LDAP (which is the access protocol for the RC in the current Globus implementation). This means, each catalogue server is implemented as an HTTP server. HTTP also allows to include the security and authentication into the request and can guarantee a secure control data transfer. The security issue will not be addressed in this paper but we point out that our approach is easily extensible to include a Grid security system, i.e. based on the Globus security infrastructure.

6.3.3 HTTP Client-Server Interaction

The main function of the catalogue server is to respond to client requests with physical file information, i.e. the exact location at a specific storage device. Client requests are formulated as standard HTTP requests (version 1.0 or 1.1).

The catalogue server supports a limited set of possible HTTP client requests: only the commands GET and HEAD are implemented. The requests OPTIONS, POST, PUT, DELETE, TRACE and CONNECT are refused by the server and the HTTP response "400 Bad Request" is returned. This is in-line with the HTTP standard [38].

HTTP provides several client-server communication commands where the basic ones for retrieving information from an HTTP server are GET and HEAD. A more flexible mechanism is URL encoded HTTP that enables to use several arguments in the client request. Since the client only requires a limited request functionality, GET and HEAD are sufficient for lookup operations. Whereas a GET request always requires that a server sends back an entire message body, the HEAD request only needs a response header that can contain all necessary information a client needs for getting the file location. It also seems natural since a client is not requesting a file but file information. Consequently, the HEAD command is used for the client lookup request.

The actual location lookup step is done as follows. A client requests a file location by issuing a HEAD request to the catServer in the following format:

```
HEAD file1 HTTP/1.1
User-Agent: catClient 1.0
Host: flora02.slac.stanford.edu
```

Note that HTTP/1.0 only requires the first line of the client request whereas HTTP/1.1 requires more client information. The server implements the HTTP/1.1 standard and thus supports both HTTP/1.0 and HTTP/1.1 client requests. The server simply responds with

```
HTTP/1.1 303 See Other
Date: Thu Mar  8 10:55:58 PST 2001
Server: catServer 1.0
Location: gridftp://flora33.slac.stanford.edu/tmp/file1
```

and thus returns the physical file location with a redirection response (303). If the file is not found, the response "404 Not Found" is returned.

Besides location information of a single logical file, a client can retrieve the entire catalogue from the catServer by using the GET method as follows:

```
GET / HTTP/1.0
```

and the server returns the entire file catalogue to the client by starting with a "200 Ok" response. Note that only the request for "/" is allowed in the GET method. Any other file request in GET results in a "403 Forbidden" response from the server. Sophisticated file

filtering is not supported and only entire file catalogues are returned rather than subsets of a file catalogue.

Although the server is multi-threaded, too many requests can arrive at the same time and the server may not be able to handle another concurrent request. In such a case, the server responds with a message that it is currently not available, and the client has to retry later. The HTTP response 503 (Service Unavailable) is used to indicate that the catalogue service is temporarily unavailable and the client can try again after a time in seconds specified by Retry-After. An example for the server response:

```
HTTP/1.1 503 Service Unavailable
Date: Wed May  9 13:26:32 2001
Server: catServer 1.0
Retry-After: 60
```

6.3.4 Grid Lookup Protocol - GLP

As pointed out, in the replica catalogue only the site information of a physical file is stored and not the entire physical filename. Each time a physical filename is required, the URL-map step is needed and the corresponding catalogue server holding a copy of the requested file needs to be queried for the exact physical file information. Thus, in the replica catalogue the physical location of a file is the address of a catalogue server at the particular site. In detail, the URL constructor in the Globus replica catalogue needs to store the host name and the port of the catalogue server of a particular site. Note that this is the only modification that needs to be done in the Globus replica catalogue. Since there is not real limitation for inserting URL constructor information into to replica catalogue, it is easy to support the distributed approach pointed out here. As a result, the location object in the replica catalogue only identifies the site but not the exact location of the file within the site. Finally, the proposed approach does not make any assumption about the distribution/replication of the global replica catalogue.

A convenient way to tell a client that another lookup step is required is to introduce a new protocol name. We call the the *Grid Lookup Protocol (GLP)*. GLP requires the use of an HTTP HEAD request to the server specified after the protocol name. In addition, the client has to provide the logical filename that is used in the LDAP request to the Globus replica catalogue. A file request scenario using GLP hence looks as follows.

1. client sends an LDAP request including a logical filename to the replica catalogue server (e.g. **dir1/file1**).
2. the replica catalogue server looks up the logical filename and returns the site information including GLP (e.g. **glp://catserver1.cern.ch:3003**)
3. the client recognises GLP and sends a HEAD request to the catalogue server (e.g. to catserver1.cern.ch at port 3003, **HEAD dir1/file1 HTTP/1.0**)
4. the catalogue server returns the URL of the requested file (e.g. **gridftp://machine1.cern.ch/data/dir1/file1**)

6.4 Implementation Details and General Remarks

In the following section we describe some implementation details of the software and ways how we achieve good lookup performance for the catalogue server. Furthermore, we discuss general replica catalogue issues and the possible usage.

6.4.1 Server-side Caching

Given the fact that a file catalogue in a Data Grid will contain millions of file entries, a lookup operation on the catalogue can be rather time consuming. We assume that an optimal index data structure is used for finding and accessing catalogue information quickly. However, even if efficient indexes are used, a catalogue lookup might still be relatively slow due to the fact that lots of information needs to be queried and the catalogue might be too big to be kept entirely in main memory. Thus, a possible index has to take into account disk I/O. Note that such indexing methods have to be done together with the storage system and the catalogue implementation (e.g. the catalogue is stored in Oracle and a native Oracle index can be used) and are not easily possible for flat file catalogues.

In order to have increased performance on local catalogue lookups regardless of the underlying index used in the replica catalogue storage system, the catalogue server caches single user requests. The cache is implemented as a hash function with a timeout parameter. For every client request, the server first looks up the requested value in the server cache. In case of a cache miss, the server searches for the requested file in the file catalogue and copies the result into the cache. On inserting a new value into the cache, a timeout value is assigned to the entry. Thus, the cache refresh policy is a modified LRU policy where entries expire after a certain amount of time.

In principle, the cache size grows when a certain limit is reached and more memory is allocated. In order to have a reasonable proportion between cache size, cache hit rate and server response time, the timeout value has to be adjusted to the application needs.

6.4.2 Logical Filenames

Logical filenames inherently exhibit replacement idiosyncrasy. This is because when the object that a logical filename points to is replaced without changing the logical name, there is no indication that there was temporal change in the meaning of the name. This has advantages as well as disadvantages.

The idiosyncrasy allows physical objects to be moved and even renamed without any need to change the logical filename. This simplifies load balancing and system recovery. On the other hand, should an object actually change and the associated logical name is not changed or, more likely, given a previously valid name; it is possible that the wrong data will be fetched. This actually happens frequently in systems that cache or abstract logical names into other domains.

The solution to this problem is well known. Each object mapping (i.e., logical to physical name) is assigned a permanently unique identifier (e.g., a DCE [30] UUID). Only the unique identifier is used when looking up a mapped object in a secondary catalogue. Thus, logical names must be cached or abstracted along with their associated unique identifiers. This requirement allows a client to immediately discover incompatible object changes since the new object will not be found using the old unique identifier because the changed mapping will be necessarily assigned a different unique identifier.

Our catalogue protocol naturally accommodates a universally unique identifier (UUID). This is because the UUID can be added to the logical name without upsetting any visible naming conventions by using the standard '?' URL constructor. For instance, the master replica catalogue would map a logical name to a URL that contains the UUID. A response of

```
glp://cs.slac.stanford.edu/objy/db/file?uuid=ed7f5bf8-44e6-
11d5-8c26-31fd864f186d
```

could be the result of one such mapping. Subsequently, the local catalogue would simply

map the URL containing the UUID to the actual URL to be used to fetch the file. This allows each site to optionally and independently implement UUID matching, as needed.

6.4.3 Redirection of File Transfer Requests

In the example in Section 6.3.1 we considered a file transfer request to get a file from a particular resource. The URL map step is done before the actual file transfer is initiated. There is another possible approach to have the URL-map step directly in the FTP server. Thus, an FTP client application only needs to issue a GET command to an FTP server which then executes the URL-map step prior to actually starting the file transfer. The FTP server can use the proposed approach to receive the physical file information from the catalogue server. This approach requires the introduction of a specialised FTP server and is currently under consideration in the Globus GridFTP effort.

6.5 Catalogue Re-synchronisation

6.5.1 The Stale Data Problem

Like in any distributed system, the possibility of errors and failures is much higher than in a single, stand-alone system. One of the main problems that can occur in our distributed catalogue environment it that servers are not available or catalogues are not up-to-date. Server down times can be improved by replicating services and providing hot-failure in case one server is not available. We do not address this issue here but are more concerned with the freshness and the correctness of data in the distributed catalogues.

In a Data Grid (in particular in High Energy Physics) we assume that most data is read-only and thus files registered in the replica catalogue normally are not changed anymore. This is a valid assumption that is also used in the Globus replica catalogue. However, we consider the fact that files are moved from one storage location to another and hence the file storage location (i.e. the physical filename) needs to be updated. Furthermore, files can just be deleted and the replica catalogue information needs to be kept up-to-date.

We assume that the default operations on file and replica catalogues are done via specific Grid tools that update the file system content (i.e. new files are created or deleted) as well as the replica or file catalogue content. In other words, if a new file is made available to the file catalogue, a particular registration tool registers the file entry in the file catalogue. The same is true for the deletion of files. However, since in a Grid environment we cannot force users to always use Grid tools for correctly operating on files registered in the Grid, there is the possibility that conventional UNIX tools like move or delete are applied to registered files. In such a case, the catalogue information is not correct anymore. To re-synchronise the catalogue with the actual physical information, a separate service monitors with a particular interval the physical storage locations of all files in the local file catalogue. In other words, the service checks if the file is available at the location stated in the catalogue. Consequently, stale information in the catalogue is removed.

A simplifying assumption is that stale data can be kept in the file catalogue until the data gets actually accessed but we do not want to have this as a default option. Consequently, a scheduled monitoring and checking is the default, and clients that detect stale data can send update information to the re-synchronisation service.

It is interesting to note that this "stale data problem" normally does not occur in a distributed database since the database management system has full control over read/write access to data.

6.5.2 Re-synchronisation Implementation

The re-synchronisation service is implemented by a separate re-sync server. The time interval when to check the existence of files can be specified explicitly. By default, the server runs permanently as a background process. If a user detects stale file information, it can send a stale file request to the server by indicating the exact file location. The server starts the file check procedure and cross checks if the file is indeed not at the indicated location. Once the file catalogue is updated locally, the re-synchronisation server can contact the global replica catalogue indicating that some stale information is in the replica catalogue.

Care must be taken when a mass storage system like HPSS is used since the requested file might not be available at the disk location indicated in the catalogue. Before a file entry can be deleted, it has to be checked with the mass storage system catalogue if the file is available there.

6.6 Experimental Results

In the following section we discuss the implemented software and give experimental results in a distributed Data Grid test-bed with machines distributed to two continents. The main purpose is to show that our approach scales well to a high number of entries in the catalogue and that the performance overhead for the additional HTTP lookup step is neglectable.

Note that a client request is first sent to the Globus replica catalogue which then returns a redirection response. This request is done via LDAP. If a Globus replica catalogue is used without a redirection step (the conventional usage), the client request would have been served already now. Since we propose an additional lookup step, we analyse this additional lookup step (i.e. its performance overhead compared to a single LDAP request) and do not put much attention to the performance of the LDAP lookup[1].

The first prototype has been written in C++ on Solaris 2.6, 7, and Linux RedHat 6.1 and 6.2. The catalogue server called catServer is available in version 1.0. We have implemented two catalogue interfaces for the local file catalogue. The first one is a simple, flat file approach and a second implementation uses an Oracle 8i database to store catalogue information. In the Oracle case, a single table with three attributes is used. The design of the catServer makes it easy to provide further interfaces to other storage and catalogue systems.

For the flat file interface, a file locking implementation guarantees that all transactions on the file catalogue are exclusive since a re-synchronisation process and a catalogue lookup process can execute concurrently. Consequently, all transactions are safe and do not produce dirty reads. In case of the Oracle approach, the database management system takes care of this.

The catalogue server is a multi-threaded server which allows several concurrent user requests and thus is scalable to a large Grid environment. Each site that uses data storage and wants to make the files available to the Grid through the replica catalogue has to have a catalogue server (catServer 1.0) running locally. In our test environment, four different servers have been used. One of them uses the Oracle interface (flora01) whereas all other servers have interfaces to flat file catalogues. The test environment contains machines on two continents (see Table 1 for details) and thus reflects a small scale Data Grid environment.

Each of the UNIX machines is running a catalogue server and a corresponding local file catalogue. A Globus replica catalogue is installed and running on a RedHat 6.2 Linux operating system at SLAC, California.

In order to demonstrate the effectiveness of our approach, the following requirements have to be met:

[1] For more information on LDAP performance with the Globus replica catalogue refer to [4].

machine	operating system	location
flora01	Solaris 7	SLAC, Menlo Park, CA
flora02	Solaris 7	SLAC, Menlo Park, CA
copenhagen	RedHat 6.2	SLAC, Menlo Park, CA
lugh	Solaris 7	Caltech (CACR), Pasadena, CA
suncms66	Solaris 2.6	CERN, Geneva, Switzerland

Table 6.1: Hardware used in the tests

- The overall query lookup performance of the catalogue system should not exceed the order of 5 to 10 seconds. This is feasible in a large Data Grid with several millions of files. Since not all of the files are replicated to all sites, the distribution of the catalogue system also helps to load balance the lookup for single servers.
- The system shall allow for usage of several site catalogues in the order of 100 to 1000. We achieve this by relying on the query performance of the central replica catalogue which can also be replicated. This does not affect our proposed catalogue servers that only need to serve user requests for physical file locations.
- Replica catalogue data needs to be updated: this can be achieved with external tools that operate on the local file catalogue. It is implementation specific (Oracle vs. flat file approach) and the locking and update time should not block the lookup.

From these requirements that are reflected in our catalogue design we conclude that the performance of a local catServer is rather well defined: it needs to support lookups of concurrent users and local lookup time needs to be fast. Since we provide a catalogue interface to a relational database management system (Oracle 8i) we assume that the lookup time of a catalogue query depends on the performance of the underlying database management system and the index used. A single file catalogue will have a few million entries and Oracle as well as many other database management systems support several thousand lookups per second. In our performance results we do not pay much attention to the local lookup performance (since it is backend specific) but stress the additional, negligible overhead we gain with the redirection step.

6.6.1 Cache Performance

By default, a catalogue server enables the server cache but it can be disabled with a command line option when the server is started. We tested the cache performance for the flat file catalogue because it does not use any indexing mechanism and thus the speedup gained through caching is more significant.

The local file catalogue contains 10,000 file entries and the catalogue has a file size of about 700 KB. The local lookup for a single file takes about 1990 ms. Note that our time measurements have a precision of 10 ms. We specify the real time (the time that elapses from the start until the end of the program), the user time and the system time for the lookup operation.

single client request: 1990/ 0/ 0 r/u/s [ms]

If the file is requested a second time, the server has it already in its local cache. The response time for the request is now:

machine	protocol	backend	RTT	lookup
copenhagen	LDAP	ldbm	0m	10ms
flora01	HTTP	Oracle 8i	0ms	170ms
flora02	HTTP	ASCII file	0ms	10ms
lugh	HTTP	ASCII file	25ms	30ms
suncms66	HTTP	ASCII file	165 ms	170ms

Table 6.2: Comparison RTT and remote server lookups of a single filename

single client request: 10/ 0/ 0 r/u/s [ms]

If no server cache is used, a subsequent request to the same file has the same lookup response time as the first lookup since the entire file catalogue needs to be searched in order to find the requested file. Interestingly, file system caching does not reduce the response time. Consequently, the catServer cache reduces the lookup time very significantly and hides long latency lookup operations on file catalogues. The lookup time of 10ms is very small and thus shows excellent lookup performance for cached values.

We then used a file catalogue with 100,000 entries and a file size of 7 MB. The server lookup time was about 20s but 10ms for a cache lookup.

Note that the lookup performance for the flat file implementation is constant for any lookup operation, no matter if the entry is found at the beginning or the end of the file. One would expect a linear search behaviour but since a logical file can have multiple physical files, the entire file has to be searched.

6.6.2 Wide-area Lookup Performance

A client request always starts with an LDAP catalogue lookup. In our test environment, we use an OpenLDAP server version 1.2.7 and the OpenLDAP ldbm database back-end on Linux. The LDAP backend only contains a few entries and the lookup request is served within a 10ms time frame.

Next we analyse the wide-area behaviour of HTTP lookup requests. A catServer is installed at suncms66.cern.ch at CERN in Geneva, Switzerland. The round-trip-time (RTT) of this wide-area link is between 162 and 165ms. The catalogue now only contains 10 file entries and is locally browsed in less than 10 ms. A remote lookup request is served by suncms66 in about 170ms which shows that the RTT is the most significant performance parameter for the lookup request. The catalogue lookup operation time is the same as for a pure local lookup and only the RTT has to be added. Table 2 shows RTT and lookup response times for different local and remote servers. The client is running on flora02.

Since the local lookup time is the same at every site, the catalogues only contain 10 items in each of the sites. Note flora01 uses the Oracle interface and contacts an Oracle server for each lookup request. This is another local client-sever communication step that also contains the client authentication. The power of a relational database backend and the usage of a database index is only visible for a large number of file entries. A detailed query performance analysis is beyond the scope of this thesis.

6.7 Conclusion and Future Work

We have presented a novel distributed replica catalogue approach that allows for site autonomy and reduced update operations on a central replica catalogue when files are moved within a

site. Site automonomy provides administrative scalability while reduced update operations provide performance scalability when in a volatile environment. HTTP redirection is used for catalogue lookups. Local file catalogues are re-synchronised by a special re-synchronisation server to provide freshness and consistence of file and replica catalogue data. The degree of up-to-date information in the catalogue can be specified by the application user. Our experimental results show that the proposed approach is a viable solution to the replica catalogue problem in a world-wide distributed Data Grid.

The ideas of our distributed replica catalogue have an important influence on the architecture of the replica catalogue system in the European DataGrid project [33]. Catalogue redirection, distribution and replication are now essential features in the design and we continue our research and development work within DataGrid as well as in co-operation with the Globus project.

Chapter 7

Replica Update Synchronisation

In the previous few chapters we presented replication work that deals with read-only file replication. Now we go a step further and allow for updates in some replicated data files. In this chapter, we first motivate file update replication in a Data Grid and discuss different approaches. Since replicated data needs to be consistent at several sites, we discuss data consistency issues, propose a possible consistency service and different replica update propagation protocols.

7.1 Update Issues for Replicated Files

Once replicated files are allowed to be updated, two or more replicas do not have the same content for some time. In other words, they are not consistent anymore. A separate update synchronisation process needs to take care of remote updates and keeping replicas consistent. Before we go into more details, we give a motivating example where replica update synchronisation is required and in which form.

7.1.1 A Motivating Example

For illustration, we assume a Data Grid with four sites distributed to Europe and the United States. The four sites are CERN (Geneva, Switzerland), Bologna (Italy), Lyon (France) and Fermilab (Chicago, Illinois, USA). This corresponds to the regional centre model introduced in Chapter 3. We further assume that all these three sites have databases and CPU resources locally, the sites are connected via wide-area network links and use Grid middleware software for e.g. authentication, file replication and possible job scheduling. Since each of the sites has its own independent database management system, new data can be produced at each site rather independently. When new data is produced, data is stored in files which are then registered in a Grid wide replica catalogue using logical and physical filenames.

In principle, we can distinguish two kinds of possible updates:

- creation of new files: only the replica catalogue information needs to be updated
- update of existing files: updates need to be propagated to remote sites and then the replica catalogue information has to be adapted too (e.g. time stamps need to be changed).

Whereas in the first case, only the replica catalogue system needs to deal with updates and changes, the second case is the traditional replica update scenario which we discuss in detail in this chapter. As an example assume that a set of files is fully replicated to all four sites. Now, assume that a database file called `file1.dbf` (which is fully replicated to CERN,

Bologna, Lyon and Fermilab and also registered in the replica catalogue) is updated at CERN. The local database management system makes sure that the file update is done in a consistent way. However, a user at Fermilab still has the old version of the file and does not even know about possible changes to `file1.dbf` at CERN.

A replication software system is required that, firstly, notifies remote sites that updates of files are done and, secondly, propagates and applies file updates also at remote sites. Based on the importance of the file as well as other possible policies and constraints, updates can be made visible to remote sites either immediately or with a certain delay. Immediate update corresponds to synchronous replication and normally has to be achieved by the distributed database management system within an update transaction (see Section 2.3). Delayed updates (asynchronous replication) can take place at any point in time after an update has been done locally. Based on our example above, the site administrator and database manager at Fermilab has decided that updates are checked for each hour or day and thus are not visible immediately. We define this as the pull model.

Definition: In a *pull model* the receiver of data decides when to get data and "pulls" it from the remote site.

On the other hand, the database administrator at CERN may decide to log changes locally and propagate the changes after a certain time interval in batch mode. Thus, CERN needs to have write access (or delegate write access) to all remote sites that have replicas of the currently updated files.

Definition: In a *push model* a data production site sends ("pushes") files or updates to a remote site without active interaction of a remote site.

Both models, push and pull, have advantages (immediate update, etc.) and disadvantages (security, space availability, etc.) but in this chapter we assume the pull model and that a site has full write access to remote sites holding replicas. The pull model is also the way object replication works in most distributed database management systems.

To finish the example, after a batch update replication process has been started, a particular file update method (see Section 7.1.3) has to actually update the file or the content. In order to do this successfully, it has to be guaranteed that the updated file is not currently read by another process. In other words, either the update process needs to be isolated in order to avoid dirty reads of concurrent readers or a write lock needs to be granted on the file that is only gained if no current read nor write operation is done on the file. Further details are given in Section 7.1.3.

7.1.2 Infrastructure Required for Replication Middleware

As identified in Chapter 3, Grid replication middleware is used for the actual replication process. Whereas pure file replication uses transfer mechanisms like GridFTP, we have seen that higher-level replication tools like GDMP require notification and thus message passing for co-ordination of remote replicas and thus remote sites. For update synchronisation we assume a more complex message passing system and define it as a *transaction system* for implementing update protocols. Every time a client wants to update data, the transaction system has to keep track of which files are changed, update a replica catalogue, notify remote sites and actually update remote replicas.

Transaction System

As compared to a traditional transaction system in a database management system, the middleware transaction system will not satisfy all four ACID[1] [28] principles, neither allows for read and write operations within a transaction, but has to satisfy some message passing, locking, restart and fault tolerance features. Whereas a classical database transaction is executed locally within a single, central database management system, a distributed system, i.e. the Grid replication middleware, requires a transaction system that spans several sites in the Data Grid.

Definition : A *local transaction* is done by the database management system at the local site whereas a *global transaction* is an inter-site transaction and spans multiple sites and thus multiple database management systems. Furthermore, *local consistency* is maintained by the local DBMS whereas *global consistency* spans multiple sites in the Data Grid.

In other words, global transactions are used to synchronise "independent" databases located at different remote sites. The database management system is supposed to be a central and not a distributed one, and does not know about remote sites and thus is supposed to be independent. This also allows for usage of any file or data format locally. More information on the replica transaction system and update replication protocols is given in Section 7.1.3.

Concurrency Control

Once a replica can be updated, concurrency control issues have to be handled. Whereas a database management system provides a powerful locking mechanism as part of the transaction system, file systems normally only provide advisory locks to files. Thus, if a file is updated at one place, there is no guarantee that a concurrent reader does not read stale data. This is even more true for a replication middleware: if replicas located in a distributed file system are opened by several processes on different machines, it is nearly impossible to get hold of all currently available file handles and thus file locks.

For the discussion in this chapter we assume that a Storage Element (as defined in Chapter 2) provides a *lock server* that keeps track of file locks and assigns file handles to applications. This simplifying assumption holds for mass storage systems that have full control to all files within their name space and thus is also valid for the discussion here. File locking is most important when files are replaced or updated by the consistency service - write locks are needed. Thus, in order to have a functional and properly working consistency service, applications need to gain read locks when opening files for read access.

Replica Catalogue Updates

For read-only files, a replica catalogue normally only holds static file attributes for logical or physical filenames. In an asynchronous replication environment, replicas of the same physical file might have different values and thus require different time stamps, i.e. modification dates. Consequently, the transaction system needs to update replica catalogue information, too. If a logical file is requested by a Grid user, the Replica Optimiser (see Chapter 8) needs to take into account the two different dates and thus versions of the file.

7.1.3 Methods for File Replica Updates

In Section 2.4.1 rather complex replica update mechanisms for distributed databases are outlined. In this sub-section, we introduce three simpler update mechanisms for file replicas and

[1] ACID is an acronym for Atomicity, Consistency, Isolation, Durability

present the specific update propagation protocols[2] and detailed discussions on protocol complexity and efficiency in the following sections.

We reuse the example from Section 7.1.1 to explain the update methods. For simplicity, it is assumed that all files residing at CERN are master replicas (i.e. can be updated by an update process) and all other replicas are secondary copies. As illustrated in Figure 4.1, each site in the Data Grid using file update replication needs to have a consistency service (details in the next section) running locally in addition to the database management system.

Simple File Replacement

We start off with the simplest case of file replica updates: file replacement. In detail, if a file is successfully updated at one site, the entire new file is sent to remote sites where replicas of this file reside and old replicas are replaced by the entire new file.

As an example consider that `file1.dbf` is updated by a database transaction of a local database management system at CERN. When the update transaction is finished and the file is not touched anymore by a write operation, the file is ready to be transferred to all sites holding replicas of this file. The local consistency service has to be notified that `file1.dbf` is updated and then takes care of the update propagation process and concurrency issues at remote sites. We further assume that CERN triggers the update process immediately to remote sites as opposed to have a batch replication every few minutes, hours or days.

To conclude, the only precondition to this replication method is that the file is not allowed to be written anymore when the update propagation process takes place. The local database management system is not aware of remote replicas and can do updates independently of other replicas. In more detail, a person at CERN can use a database application program that updates the database file and then calls a particular method or command line tool of the consistency service which in turn takes care of the update propagation process. This is a totally asynchronous replication approach and is applicable to all the three update methods presented here.

The *File Replacement Protocol* presented in Section 7.3 implements the file replacement method.

Binary Difference Between Replicas

Whereas in the previous approach entire files are updated, a common replication strategy is to propagate only the difference between an old and the new version of a data item or file. A method for detecting these differences, i.e. changes from one version to another, is required. Whereas a common approach is to store a log of changes (see next method), the "binary difference" only compares entire files: old and new version of the same file. In principle, we can distinguish between two kinds of files: ASCII files and binary files where binary files are all other files that do not store file contents in ASCII formats [125].

There exist several methods for detecting the difference between two text files but we need a generic approach that works for all kinds of files, i.e. ASCII and binary files. In particular, database files are in binary format and thus require a binary difference approach. Consequently, we use a tool called XDelta [79] that produces the difference between any two files (binary or ASCII). This difference can be sent to the remote site. This site can then update the file, which is out of date, by merging the diff file with the original data file. XDelta is a library interface and application program designed to compute changes between files. These changes (deltas) are similar to the output of the Unix "diff" program in that they may be used to store

[2]Note the difference in the usage of the term "protocol". Whereas in the Grid community, the term protocol is mostly used to identify a network protocol like HTTP or FTP, in replication research a protocol is the form of interaction between replication servers in order to update replicas.

and transmit only the changes between files. However, unlike diff, the output of XDelta is not expressed in a human-readable format - XDelta can also apply these deltas to a copy of the original file(s).

In contrast to the previous method, before a change on a file is done, the original file needs to be restored and a separate copy has to be created for comparison of changes. For instance, if `file1.dbf` is supposed to be updated, the consistency service needs to be contacted and then creates a copy of the file, e.g. `file1.dbf.copy`. Once the updates on `file1.dbf` are done, the consistency service uses XDelta to find the differences between `file1.dbf` and `file1.dbf.copy`, and sends these differences to all remote sites holding replicas of `file1.dbf`. At the remote sites, the consistency service is then again responsible for updating replicas.

The *Binary Difference Protocol* presented in Section 7.3 implements method presented here.

Advanced Method Based on File Logs

In the previous two methods no knowledge about the internal structure of files is required and thus they are very generic and applicable to any possible data store using files. However, update performance might not be optimal since knowledge about data structures or file formats is not used. Furthermore, designing a data model with persistent objects[3] that are already aware of replication seems to be more optimal in terms of update replication performance. Note that we are still operating at the middleware layer and do not have influence on the internals of a database management system (we cannot change transactions in a commercial database system where we do not have the source code). However, the data model including all data manipulation methods needs to be aware of replication. In particular, when a change is done on one object, the knowledge about what has been changed and where other replicas reside needs to be available.

Since most of the applications in the DataGrid project use object-oriented programming languages and thus also an object-oriented programming interface to a database management system or object data store, we assume a data model that uses such an object-oriented interface. The underlying data store can either be a pure object-oriented database, a relational database with an object-oriented interface or simpler object stores than full grown database management systems.

In [117] we proposed an object-orient approach where an object creation method has to take care of creating object replicas at remote sites. A replicator object is then responsible for propagating updates. We now present a slightly simpler approach which relies on other Grid services like the replica catalogue and the file copier service (see Section 4.1). Since the Grid services presented in this thesis and developed in several existing Grid projects deal with files rather than objects, an additional object-to-file mapping step is required.

Definition: An *object-to-file mapping* procedure maps a persistent object to a file where it is stored on disk. In principle, a logical object can be mapped to a logical file and a file replica catalogue then resolves the physical file(s) where the object(s) reside(s).

Consequently, update methods on persistent objects have to be provided explicitly for each replicated object where changes have to be asynchronously propagated to remote sites holding replicas of this object. Within the bodies of the update methods, the consistency service needs to be called with the update information. Additionally, a data store specific object-to-file mapper needs to be provided that maps a single object to a file. Since every persistent object is identified by its unique object identifier (OID), the object-to-file mapper takes the OID as

[3]Persistent objects are object instances of classes (in an object-oriented programming language like C++ or Java) and stored on secondary or tertiary storage devices like disk or tape rather than main memory.

input and returns a file. By use of the replica catalogue, the consistency service can identify where replicas reside and execute the update process.

We now discuss a brief example and give details in the next sections. We assume a very simple C++ object (stored persistently) like follows[4]:

```
class Person {
public:
  void setName(string name);
  string getName();
private:
  string name_;
};
```

We assume that the object is already created and has an OID 1222-22A. In order to use the consistency service (which provides a language API) all the methods that change variables in the object need to call the consistency service which then logs the changes and applies them to remote objects. Since the consistency service needs to reproduce the exact change, it needs to know the OID, the update method and the new data member values. Note that we define the functionality of the consistency service later in more detail but here we give a short impression of how an end-user can use such an update method. Thus, one consistency service API for updating a single object looks like follows:

```
updateObject(‘‘1222-22A’’, setName, name_, ‘‘Stockinger’’);
```

The above method has to be included into the body of the method `Person.setName`. The consistency service will then take the object-to-file mapper, which needs to be configured once for the data store, writes the update information into a log file and later updates remote sites holding replicas of the object. Internally, the consistency service uses the file copier service to securely transfer the log file (holding all update information) to remote sites. Since a (database) update transaction on the persistent data store might fail and thus the change to the object needs to be recovered, all `updateObject` calls have to be committed with a single `commitObjectUpdate` or `abortObjectUpdate` after the database transaction has committed successfully. After the update is committed, the consistency service will apply the same object updates on remote objects using the corresponding object update methods as identified in the log file.

The *Log-Based Protocol* presented in Section 7.3 implements the update method outlined here.

7.2 Consistency Service

The Consistency Service is a novel Grid service sitting on top of the Replica Manager service as we defined in [32] and thus is a high-level replication service. The main functionality of this new service is to synchronise updates on file or object replicas and keep them consistent. In this section, we discuss different consistency requirements, how they can be met and present a consistency service for update synchronisation of replicated files and objects.

[4]The class presented here is not a persistent class yet. For instance, if Objectivity is used, the class needs to inherit from a persistence capable class `ooObj`.

7.2.1 Consistency Requirements

The degree of consistency of replicas depends on the application and can have different forms. Strict consistency means that all replicas are always 100 percent in sync and thus fully consistent. Due to the locking overhead of keeping huge amounts of distributed data in sync [49], 100 percent consistency is a very impractical solution for a Grid environment. Thus, if knowledge about the data and user requirements (use cases) are available, one can relax this strict consistency requirement and allow certain parts of the data to be out of sync for a particular amount of time. For instance, a site A in a Data Grid says explicitly that newly created files at other sites B, C, and D have to be transferred to site A within two days. This means that the replica creation process can be done within a 48 hour time frame. Within this period the state of physical files can be inconsistent. Another example is that writable replicas have to be updated and synchronised every 10 minutes. As a third example, we mention that updates of meta information might be 100 percent synchronised. We derive a clear need for different consistency models and implementations.

The easiest way to tackle the consistency problem is in the field of read-only data. Since no updates of any of the replicas are done, the data is always consistent. We can state that consistency can reach its highest degree. Once updates are possible on replicas, the degree of data consistency normally has to be decreased (provided we have read and write access to data) in order to have a reasonable good response time for accessing data replicated over the WAN. Consistency also depends on the frequency of updates and the amount of data items covered by the update.

Definition : The *degree of data consistency* depends on the update frequency, amount of data items covered by the update and the expected response time of the replicated system, i.e. the Data Grid.

The highest degree of consistency can be established by having fully synchronous replicas. This means that each local database transaction needs to get acknowledgements from other replicas (or at least a majority of replicas). In practice, as well as in the database research community, this is gained by global transactions which span multiple sites. As discussed in Chapter 2, high consistency in synchronous replication approaches comes at the cost of relatively worse performance for global writes compared to local writes with no replication at all. One has to decide carefully if the application requires such a high degree of consistency.

Based on this trade-off between consistency and response time, the type of replication protocol and hence the data consistency model have to be well adapted to the application. This is also pointed out in the current database literature [17]. Data in a Data Grid will have different types and not all of them require the same consistency level. Furthermore, we claim that a replication system of a Data Grid should not only offer a single replication (and consistency) policy but several ones which satisfy the need of having different data types and degrees of consistency.

For a middleware replication system it is rather difficult to provide this high degree of consistency since global, synchronous transactions are difficult to establish (compared to a pure DBMS approach): within a DBMS a global, synchronous transaction can be an extension of a conventional, local transaction, i.e. the DBMS specific locking and transaction mechanism is extended to remote replicas. Since a distributed DBMS like Objectivity has built-in global transactions, no additional communication mechanism like low level sockets or a higher level message passing library is required on top of the existing DBMS. Hence, the performance for an integrated distributed DBMS is superior to middleware replication systems that have to use external communication mechanisms.

Since current distributed database systems do not provide flexible consistency levels for different kinds of data, we aim for a *hybrid solution* where a local site stores data in a DBMS or persistent data store which also handles consistency locally by managing all database transactions locally. Grid middleware is required to provide communication and co-ordination between the local sites.

7.2.2 Levels of Consistency

Different levels of consistency can be gained by allowing *delay* or *latency* in the update propagation of changes to remote replicas. Whereas immediate propagation has low latency and thus high consistency, asynchronous or batch replication collects a set of changes and propagates them at a certain point in time to other replicas.

Definition: *Consistency levels* are used for a set of identical replicas and can be expressed by the time delay for keeping replicas identical.

Delays can take place at either the originator (the producer) of an update or the receiver (remote replicas). In principle, at each site a consistency service needs to be in place that can be configured to allow for levels of consistency. A producer site may want to notify remote sites immediately after an update has been done locally and automatically trigger the update propagation process or might want to update remote sites at fixed times like every hour or every day. The same is true for the receiver of an update notification message: the updates can be applied immediately or with some delay.

An important aspect in a Data Grid is that several different data types exist and several levels of consistency are required. In particular, a replication model should be chosen by the user and has to be adapted to the data model. For instance, small files holding a few objects might be synchronised with a very low latency whereas large files with not very frequently accessed data need to be up to date only after a few hours. In addition, consistency restrictions are imposed by the file size, the number of replicas and network conditions (fault tolerance issues).

All the models presented above are automatic update replication models where hardly any user interaction (except service configuration) is required. In some environments, a semi-automatic system is preferable where a site administrator gets notified about certain updates and changes and then can decide on demand if and when the updates shall be applied to the local data store.

To conclude, the consistency service has to provide several technically possible solutions for replica update synchronisation. The "best suitable" policy then has to be adapted to the user environment and might only be gained after some tuning of parameters. It shall provide general consistency levels and interfaces but the actual service implementation is tightly coupled with the underlying data store and data model. Whereas in a relational DBMS a standard query interface like SQL can be used to handle updates at a local site and propagate them efficiently to all replicas, a data store like Objectivity does not provide such a high-level interface and thus requires more sophisticated update mechanisms. We provide a detailed discussion on consistency models in [32]. Further ideas on consistency and delays for replica updates can be found in [104].

7.2.3 Architecture

We outlined several requirements and consistency issues and now point out how a consistency service can achieve the requirements and which architectural components and interactions are necessary. In principle, several file and object update methods have to be provided as discussed

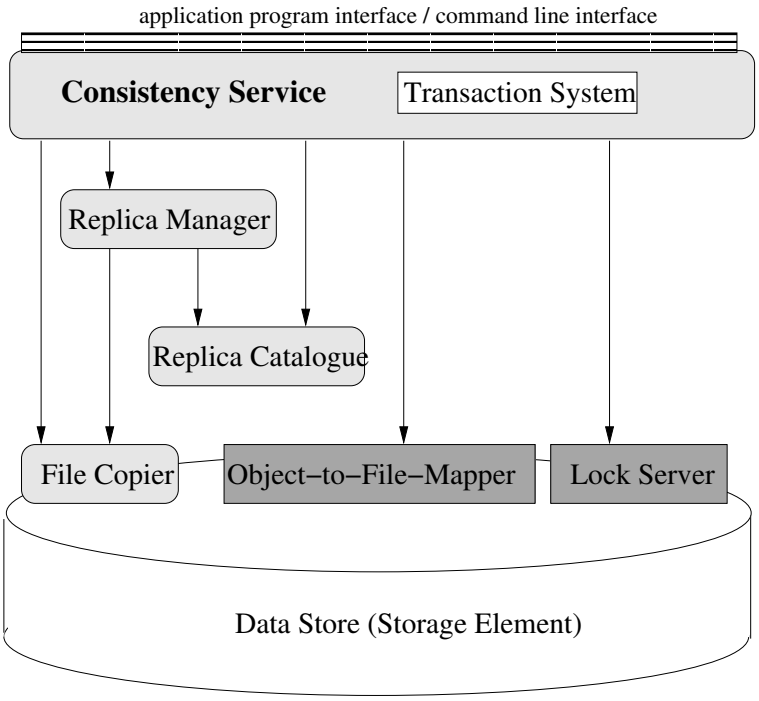


Figure 7.1: Architecture of the Consistency Service and interaction with other Grid services and data store specific components.

in the previous section. It should be easy to plug in new implementations for other possible update methods.

The basis architecture of the Consistency Service is a layered one that is built on top of some existing Grid services like the

- Replica Manager: for creation of file replicas at remote sites; transferring files efficiently and registering them in a replica catalogue
- Replica Catalogue: for searching for logical and physical filenames and storage locations as well as storing replica information
- File Copier: for transferring data files as well as log files

In Figure 7.1 these three general Grid services plus our proposed Consistency Service are illustrated in light grey. Arrows indicate that one service is using another one. Note that the layering does not imply that one service can only use a service at the next layer: in our presented architecture, the Consistency Service can also have direct access to the File Copier and Replica Catalogue service.

Update propagation and replica synchronisation are executed on the data store (Storage Element) which can have any file or object format for storing data persistently. Possible storage technologies are object-orient database management systems (e.g. Objectivity), object-relational or pure relational database management systems (e.g. Oracle) or persistent object stores (e.g. ROOT). Since update synchronisation needs to prevent users from update anomalies and "dirty reads", a *lock server* managing all updateable files in a Storage Element needs to be present at each site. The lock server as well as the *Object-to-File-Mapper* are in dark grey since they do not represent Grid services but components that are very specific to the underlying data store and storage technology used. The Object-to-Filer-Mapper is responsible

for mapping an object (based on its unique object ID) to a logical file and the replica catalogue service then resolves in which physical file replicas of this object reside. The Object-to-File-Mapper is different for every storage technology using persistent objects and should be easily replaceable for different storage technologies. Clearly, the Object-to-File-Mapper is only used by the Consistency Service if an object replication method is used rather than file replication.

The entire Consistency Service is a client-server system where a single consistency server runs on each Storage Element. Several consistency servers need to communicate with one another in order to exchange replica update synchronisation messages and update transactions. The *Transaction System* in the Consistency Service is responsible for wide-area communication and global transactions between consistency servers. In particular, a server needs to keep track of all communication messages (transactions) with remote consistency servers, and keep a transaction table for storing acknowledgements to transactions for fault tolerance reasons. If messages get lost or do not arrive in time, time-outs and retries are provided.

Applications and in particular update programs interact with the Consistency Service either through an application program interface expressed in a programming language like C++ or command line interfaces can be used. This is true for the three update methods presented in the previous section. Details about the update interfaces and possible replication protocols can be found in the next section. In addition to the actual update replication interfaces, the Consistency Service provides a configuration interface where consistency models can be set up. In particular, a consistency server can be configured to propagate changes immediately or with a certain delay (see Section 7.2.2).

Internally, the Consistency Service contains three major components that support different update propagation protocols:

- *File Difference Manager*: It is the task of the File Difference Manager to create the difference (delta) between two files and later update another file based on the result of the differences. In our tests the XDelta software system was used but can be replaced by any other software system capable of producing a binary difference between two files.

- *Log Manager*: The Log Manager has to record (log) all changes done on persistent objects in one data store and make them available for reproducing the same changes on persistent, replicated objects in another data store. The Log Manager needs to be tightly coupled to the underlying data store and data model.

- *Object Update Manager*: The Object Update Manager is responsible for reading log information about updated objects and applying these changes to replicas. The Object Update Manager needs to know about the entire schema (for instance class definition including update methods or SQL data definitions) of possible object types to be updated.

7.3 Replica Update Propagation Protocols

We have outlined three different update methods in the previous sections and now give details on how these have to be implemented. We propose three novel update propagation protocols with increased complexity and efficiency as regards network utilisation using Grid services and data store specific components depicted in Figure 7.1. Each protocol discussion starts with a user interface (API) to the consistency service and how the consistency servers interact to update replicated data.

Each of the APIs is presented in C++ and represents a client call to a consistency server. Files need to be unique within the Data Grid and thus physical files are represented by their

physical filenames on disk[5]. Physical files can only be updated by using physical filenames in the APIs. For all the following APIs, `PFN` is a string satisfying the syntax defined in Section 4.1.2. Furthermore, all physical filenames have to be registered in a replica catalogue service.

Although three independent protocols are available, a local consistency service needs to be configured to allow for certain levels of consistency and thus the update policy. The configuration includes:

- when are updates propagated: immediately or with a certain amount of delay to be specified by administrators of the consistency service
- when are updates applied at remote sites (immediately or delayed).

Finally, also a push or pull model needs to be configured.
Before we go into details, we point out some assumptions for the Data Grid environment:

1. For each physical filename there exists exactly one logical filename. Multiple logical filenames per physical file are not allowed.
2. Before a file can be replaced or updated, a write lock needs to be gained which is managed by a separate lock server. In order to notify remote users of file locks, a file lock flag also needs to be visible in the replica catalogue. Thus, a file lock needs to be atomic and includes a lock in the lock table of the lock server as well and entry for a physical file attribute for the corresponding physical file. When we mention write lock in the protocols below, it is assumed that both lock entries are done.

   Note that this is a severe restriction compared to a database management system or a distributed file system. Whereas in the latter case, a file handle might not be assigned in case of locks (although advisory), there is no straight forward way to reject file handles in case of a lock[6].
3. For each physical file that is writable, a modification date needs to be stored in the replica catalogue as a physical file attribute. The modification date is required to identify which file is most up-to-date[7].

7.3.1 File Replacement Protocol

For the file replacement protocol, two different user APIs are provided:

- `replicateNewPhysicalFile(PFN pfn)`
  This is the simplest file update API where it is assumed that an application has updated the physical file (`pfn`) and the new physical file then needs to replace all replicas of that file. Thus, the consistency service does not need to assign any file lock on the local physical file since it is assumed that a correct update has been done already by some external program (using write locks on the file). After the method has been called, the file modification time for the physical file is updated in the replica catalogue. In order to replace remote replicas, write locks need to be gained.

[5]Note that for simplicity we only distinguish between logical filenames and disk resident physical filenames. We do not take into account that mass storage systems have further internal logical and physical filenames but our interfaces can be easily be extended to such systems.

[6]A possible way is do introduce file locks to FTP server for remote file access. However, it does not seem to be a very clean solution since local file access is still possible.

[7]Note that synchronising clocks is a well known problem in a distributed system but we assume that file modification dates are always assigned by sites holding the master of replicas.

- `replaceAndReplicatePhysicalFile(PFN oldPfn, PFN newPfn)`
  A new file has been written by an external application and is available at location `newPfn`. This method copies the file from `newPfn` to `oldPfn` (i.e. replaces the old file locally by the new file), updates the replica catalogue and then replaces all remote replicas. In contrast to the previous method, a write lock also needs to be gained locally before the file can be replaced. Once the file is replaced properly at the local site, the file is deleted at location `newPfn`.

Next, we present the File Replacement Protocol for the first of the two possible cases, `replicateNewPhysicalFile(PFN pfn)`. We assume that such a request is sent to the server and now show all the protocols steps required to update a replicated file. The local server (producer or originator of updates) is the consistency server that has received the request and also the data store is local to this server, i.e. at the same Storage Element. A remote server (consumer of the update) is a located at a different site in the Data Grid and has a wide-area network connection.

local server:

1. *check if the PFN exists in the replica catalogue*
   if yes, update the PFN attribute with a new modification time, else exit

2. *determine all replicas of PFN*
   The following replica catalogue methods are used :
   `LFN = ReplicaCatalogue.getLFN(PFN)`
   `PFNList = ReplicaCatalogue.getPFNs(LFN)`

3. *determine remote sites*
   extract the location of the remote consistency servers (remote sites) from the host name in the PFN: assume that all consistency servers listen on a the same predefined port

4. *check update policy*
   update policy determines the delay of messages/updates and the push/pull model

5. *send notification*
   notify remote server(s) about new replicas including the modification time attribute

remote server:

1. *receive file update request*
   the server receives a file update request transaction with the required information for the File Replacement Protocol

2. *check update policy*
   further actions and delay depend on the update policy

3. *get updated file from original location*
   use File Copier for the file get command and store the file under `pfn.new`: In order to keep write locks as short as possible, the new file is first stored with a new name and after the validation a write lock is gained to replace the file locally. File replacement requires only very fast file system operations (delete and rename) and thus locks the file only for a short time

4. *get file write lock for physical file*
   obtain write lock from local lock server - file lock needs to be visible in the replica catalogue too

5. *rename file locally*
   delete original file and rename the new file from `pfn.new` to `pfn`

6. *update file modification time in the replica catalogue and release write lock*

7. *delete temporary file copy*

8. *acknowledge successful file update*

For all transaction messages that are exchanged between consistency servers, acknowledgements and timeouts in case of network partitions are assumed and are not explicitly reflected in the protocol above.

For `replaceAndReplicatePhysicalFile(PFN oldPfn, PFN newPfn)` the update protocol is slightly more complex and includes an additional local write lock on the physical file and an addition copy step locally. We omit the entire protocol here due to the similarity with the one above.

7.3.2 Binary Difference Protocol

Since the Binary Difference Protocol needs to keep track of file changes, it requires two API calls that have to be used before and after an external (not under the control of a Grid service) file update process is executed. Similar to a database transaction system, a preparation and a commit phase are required. Only if both functions are successfully called in the correct order (as given below), the update propagation process can succeed.

- `prepareFileUpdate(PFN pfn)`
  The consistency service needs to know in advance which file will be updated in order to create a temporary copy first (`pfn.copy`). The `pfn` needs to point to an existing file and has to be in the replica catalogue. A file write lock is acquired. Once this command is called and a write lock is gained, an update process can be applied to the current file.

- `commitFileUpdate(PFN pfn)`
  When the update process on the file `pfn` has succeeded, this call is used to commit the changes and trigger the Consistency Service to propagate the changes to remote sites. Furthermore, the replica catalogue is updated with the new modification time of the file.

In order to provide a transaction like system for file updates, a prepared file update can also be aborted:

- `abortFileUpdate(PFN pfn)`
  The file lock on `pfn` is released and the original state of the file is restored, i.e. a possible updated new file is deleted and `pfn.copy` is renamed to `pfn`.

The entire Binary Difference Protocol looks like follows:

local server:

1. *check if PFN exists in replica catalogue*

2. *get file write lock for physical file*
   obtain write lock from local lock server

3. *create temporary copy*
   check if enough space is available for copy and create file `pfn.copy`

4. *external file update process*
   an external, user provided program is updating data in the corresponding file

5. *create diff file*
   the Difference Manager (in our tests XDelta) is used to create a file `pfn.diff` that stores the differences between the file `pfn` and `pfn.copy`

6. *determine all replicas of PFN*
   The following replica catalogue methods are used:
   `LFN = ReplicaCatalogue.getLFN(PFN)`
   `PFNList = ReplicaCatalogue.getPFNs(LFN)`

7. *determine remote sites*
   extract the location of the remote consistency servers (remote sites) from the host name in the PFN

8. *check update policy*
   update policy determines the delay of messages/updates and the push/pull model

9. *send notification*
   notify remote server(s) about new replica including the modification time attribute and the location of the diff file

remote server:

1. *receive file update request*
   the server receives a file update request transaction with the required information for the Binary Difference Protocol

2. *get diff file from remote server*
   use File Copier for getting the file `pfn.diff`

3. *get file write lock for physical file*

4. *apply changes to the physical file*
   the file `pfn.diff` is used to update the file `pfn` locally by the Difference Manager

5. *update modification time for file in replica catalogue*
   change the modification time only if the update process has succeeded

6. *release write lock on file*

7. *delete the diff file*

8. *acknowledge successful file update*

Since both API ("prepare" and "commit"/"abort") calls have to be used in an atomic way (either both of them or none), a possible problem can occur if the user defined update process does not succeed. Since a copy of the original file exists locally, a rollback operation is possible. Furthermore, timeouts can be used to notify users to manually recover from update failures. A temporary file is only deleted if the update process succeeds or a manual conflict resolution interaction has been done.

7.3.3 Log-Based Protocol

A sketch of the API has already been presented in a previous section but now we point out the exact syntax and details. We use the same example (class Person) like before.

Before an object can be updated, a Consistency Service object needs to be created and instantiated which is then passed to the actual persistent object to be updated. The Consistency Service object stores all log information about the actual update process.

- `ConsistencyService *cs = new ConsistencyService()`
  This call creates a pointer to the Consistency Service and prepares the log process for object updates.

- persistentObject.`prepareObjectUpdate(ConsistencyService *cs)`
  For each persistent class, such a method needs to be defined that passes the Consistency Service object to the persistent object. In our example, replace "persistentObject" by "Person".

- `cs.updateObject(OID, updateMethod, list of variable/value-pair)`
  Within each update method of the persistent object, this method needs to be called which logs all update information. In particular, the Log Manager will be invoked to keep track of changes. The `OID` is the unique object ID that the object needs to retrieve about itself. `updateMethod` is the name of the actual update method of the persistent object: in our example it is "setName". Since an update method can possibly change several variables, a list of variable names and their new values needs to be provided. Note that the Update Object Manager needs to know about the schema of the object. This can be done by configuring the Update Object Manager with the schema definition of the object.

- `commitObjectUpdate()`
  Whereas `cs.updateObject` stores object updates temporarily, changes of objects are written to a log file when updates on the Consistency Service object are committed. If a database management system is used, this method has to be called after the update transaction on an object that has succeeded. With this method, several updates are committed at the same time by the Object Update Manager.

- `abortObjectUpdate()`
  If updates on objects do not have to be committed to the database, updates can be aborted and the temporary log for object updates is deleted rather than written to a log file.

In this protocol it is assumed that the database management system takes care of concurrency control on local files where updates are done. However, even if a write lock is used locally, it needs to be visible to all other Grid users too (in the replica catalogue) in order to prevent them from trying to access the currently modified file. It is now up to the end user to trigger a lock on the replica catalogue entry explicitly by consulting the Object-to-File-Mapper about the file where the object to be changed is stored. Once the Object-to-File-Mapper has returned a physical filename, a write lock on this file can be made visible in the replica catalogue by explicitly calling `getWriteLock(PFN pfn)`. The complete interface for file locks is as follows:

- `getWriteLock(PFN pfn)`

- `releaseWriteLock(PFN pfn)`

- `getReadLock(PFN pfn)`

- `releaseReadLock(PFN pfn)`

The entire Log-Based Protocol looks like follows:

local server:

1. *Log Manager stores changes of objects temporarily*
   This is done each time when `updateObject` is called. At the same time:
2. *create write lock on file*
   The Object-to-File-Mapper determines the physical file where the object to be updated is located and locks the entire file.
3. *commit object updates*
   The Log Manager writes all temporarily stored update logs to a log file.
4. *determine all replicas of PFN*
   The following replica catalogue methods are used :
   `LFN = ReplicaCatalogue.getLFN(PFN)`
   `PFNList = ReplicaCatalogue.getPFNs(LFN)`
5. *determine remote sites*
   extract the location of the remote consistency servers (remote sites) from the host name in the PFN
6. *check update policy*
   update policy determines the delay of messages/updates and the push/pull model
7. *send log file*
   The Log Manager sends the log file to servers holding replicas of the files where objects are changed. For each object, the corresponding logical file is stated.

remote server:

1. *receive object update request*
   receive the log file with all changes
2. *read log file and determine files where objects need to be changed*
   use replica catalogue to determine physical files
3. *gain write locks on all physical files to be updated*
4. *Object Update Manager performs updates on persistent objects*
   extract the update methods from the log file, gain object references and update the objects
5. *update replica catalogue with new modification time*
   for each physical filename, change the file attribute “modification time”
6. *release write lock on files*
7. *acknowledge successful file update(s)*

Example

An entire example (without error handling) in C++ for the usage of the APIs for the Log-Based Protocols follows. Note that we assume that a database management system is used which assigns a handle to a persistent object. All updates on the persistent object need to be done with conventional database transactions. Once a database transaction is committed, i.e. the changes on the persistent object are made persistent, the commit method of the Consistency Service has to be called.

```
#include ‘‘database.h’’
#include ‘‘ConsistencyService.h’’
#include ‘‘Person.h’’
...
Person examplePerson;
databaseTransaction.start(); // initiate a specific database transaction
...   // get handle to a persistent object of class Person
ConsistencyService *cs = new ConsistencyService();
examplePerson.prepareObjectUpdate(cs);
examplePerson.setName(‘‘ExampleName’’);
databaseTransaction.commit();
cs.commitObjectUpdate(); // release all write locks
```

The log file contains all information from the updateObject method and the logical filenames of updated objects.

7.4 Protocol Discussion

In order to discuss the efficiency and usefulness of a replica update protocol, the following parameters have to be taken into account:

- size of the entire file: $size_{file}$
- amount of updates to be done on objects in the file: $n_{updates}$
- size of updates (in bytes): $size_{update}$
- current wide-area network throughput: f_{WAN}
- protocol complexity: how many transactions and interactions have to succeed in order to have a complete file update: $n_{transactions}$
- number of locks on the file and how long the file is locked: n_{locks}
- scalability: how many replicas exist for a single file: $n_{replicas}$
- number of concurrent users: n_{users}
- performance requirements: how fast shall the update be: $response_time$
- consistency level or update policy: $consistency_level$

The above parameters are given for single files and thus apply primarily to file update replication protocols like the File Replacement Protocol (FRP) and the Binary Difference Protocol (BDP). The Log-Based Protocol (LBP) on the other hand updates single objects

and thus might cover several files based on the object clustering. In detail, an update process may use logical object IDs and does not know in advance in which files the objects to be updated will reside. In the following discussion we analyse the second version of FRP using `replaceAndReplicatePhysicalFile` since it is the more complete update protocol and also includes locks on the local file.

Having stated the external parameters, the set of operations for replica updates (independent of the protocol used) in time order follows and will be used for the protocol comparison in Section 7.4.2. The dependency on the above stated parameters is given in brackets:

1. t_1: local update (n_{users}, n_{locks}, $n_{transactions}$)
2. t_2: notification (f_{WAN}, $n_{transactions}$)
3. t_3: remote site gets file/updates (f_{WAN}, $size_{update}$ or $size_{file}$)
4. t_4: remote update (n_{users}, n_{locks}, $size_{file}$, $size_{update}$, $n_{updates}$, n_{locks})
5. t_5: all replicas are in a consistent state expressed in $response_time$: ($n_{replicas}$, f_{WAN}, $n_{transactions}$, $consistency_level$)

Challenging factors for an update propagation protocol in a Data Grid are the number of users, files and replicas, sites and the latency for response times to access files. As an example, we assume the tier model in CERN's Data Grid with 1 tier 0 site (CERN) and 5 tier 1 sites in Europe and the U.S. Further assume that 1,000,000 files with a file size of maximum 2 GB (yields 2 Petabyte in total) are available at CERN and each tier 1 Regional Centre has about 30 percent of all files (30,000 replicas). About 1 percent out of all possible 1,000,000 are updateable. Thus, we limit the update problem to potentially 10,000 files with a maximum of 5 additional replicas in Regional Centres. Finally, roughly a maximum of 200 concurrent users accessing files is assumed where the amount of write access is very limited to under 1 percent.

7.4.1 Lock Contention and Local Updates

A major criterion for an update propagation protocol is that write locks do not block access to the entire Data Grid. For instance, in a synchronous replication protocol, if a file is updated at CERN, all replicas in the 5 Regional Centres are locked, too. Since several concurrent readers are possible at several locations in the Data Grid, it might take a long time to gain write locks and thus the update is delayed. The three proposed update propagation protocols follow an asynchronous approach and thus write lock contention on single files is only an issues within a site rather than within the entire Data Grid. This clearly comes at the cost of reduced data consistency but this seems to be preferable to no access to data. Since also remote updates only need to co-ordinate locks within a site, we conclude that the proposed protocols do not impose additional locking overhead and are thus independent of $n_{replicas}$ but only depend on local n_{users} and n_{locks} of a single replica at a time.

7.4.2 Comparison of the Three Protocols

We now start with the discussion and comparison of the three protocols FRP, DBP and LBP: The local update that takes place at time t_1 is similar for FRP and DBP but has an additional overhead for LBP since updates need to be logged. The update notification step at time t_2 is similar in all the three protocols. The file transfer process at time t_3 is very significant for the overall response time of the update. Whereas in FRP an entire files is transferred ($size_{file}$), in DBP and LBP only changes and thus smaller files are transferred over the network. The latency is rather predictable in case of FRP (assumed that f_{WAN} is predictable too) whereas

in case of DBP and LBP $size_{update}$ depends on the number of updates and the implementation of the local update procedure (to reduce the file size etc.).

The remote update procedure at time t_4 has a major influence on the overall performance of the protocol. Whereas in case of FRP the update procedure is very short (just two file system calls for delete and rename), the update process for BDP and LBP takes much more time depending on $size_{update}$ and $n_{updates}$ as well as the update performance of the File Difference Manager and the Object Update Manager.

We now state some parameters for protocol comparison:

- The overall *response time* for having all replicas in sync mainly depends on the update policy and when remote sites decide to update data. The minimum response time (no notification delay and no delay for remote update[8]) depends on all the parameters above plus the amount of sites. The minimum possible response time needs to be taken into account when providing different consistency levels. In particular, for FRP the response time mainly depends on the file size and can thus be estimated rather easily. Thus, a minimum delay for the update of a single replica can be determined and then needs to be extrapolated to several sites taking into account that shared network connections and file transfer servers are used.

  For example, assume that in the above Data Grid scenario network connections with an average transfer rate of 10 MB/s and FRP are used. Thus, for a single remote site to update a file via FRP it takes at least about 3 1/2 minutes for being consistent with the CERN site. If we further assume that the FTP server at CERN has a maximum transfer speed of 20 MB/s and all 5 Regional Centres hold replicas, a minimum of around 8 1/2 minutes is required to update all remote sites further assuming that no shared network connections are used. If we assume that only a few objects are changed in the original file and the amount of changes stored in the diff file (for BDP) or the log file (for LBP) is about 1 MB, the file transfer over the network is done within a second but the updated needs to be applied locally too. This might take another few seconds or minutes. If the number of changes gets bigger, the update time used by the File Difference Manager (for BDP) or the Object Update Manager(for LBP) increases as well and can be larger than the time it takes to transfer a file. Exact measurement and performance tuning is required to find out which protocol has a better response time for updates.

- *network utilisation*: Whereas for FRP the required throughput is $size_{file} * n_{replicas}$, for DBP and LBP it is $size_{update} * n_{replicas}$ which is normally smaller than for FRP.

- *Lock contention* and *delay* for updates of single replicas can also be compared. Here, FRP dominates over DBP and LBP. For LBP (as well FRP and BDP) locks are gained for entire files and not only for certain parts where updates are done: this might be too much lock contention on a single file as compared to a database management system where locks are applied on a set of objects, but compared to the other two protocols this lock contention is bearable.

- *protocol complexity, fault tolerance and error recovery*: The advantage of FRP is its simplicity and thus functional applicability to many environments. Since a low number of messages is exchanged and no additional components like the File Difference Manager or the Object Update Manager are required, it scales well to a large number of sites. BDP is slightly more complex than FRP, but LBP has the most complex transactions and interactions with several components where errors can occur. Since the complexity

[8] "no delay" does not mean that there is no latency in the network but refers to the configuration of the consistency service

of the protocol is direct proportional to the number of errors that can occur (messages get lost, components fail, etc.) a possible recovery process is easier for protocols with lower complexity like FRP and BDP. Note that for none of the protocols we have included error recovery nor retries in case of network partitions and lost messages.

Since LBP requires a modification in the data model, it is more prone to human and machine errors than the two file update protocols.

7.4.3 Concurrency Control and Admission Control

In the three presented protocols *concurrency control* is dealt with by introducing write locks to files. The consequence for the use of updateable files in a Data Grid is now that one needs to be aware of files that are updateable and apply read locks when a file is opened for read access. This is a significant restriction and very prone to human errors but the only way to prevent multiple readers and writers on the same file.

Admission control deals with who is allowed to access the data and which restrictions occur. Security models can be established but here we assume that users are already authenticated and discuss briefly admission control to replicas and the influence on the complexity of the update protocol.

In order to reduce the protocol complexity for updates, well defined ownership [32] limits the number of sites where updates can be applied. For a set of identical replicas, one replica is assigned master or primary copy and only this file can be updated by a local update operation. The replication system then propagates the changes to all other replicas. The replica catalogue stores the information that a replica is a master as a physical file attribute. For all our protocol discussions above we have assumed that there exists only one master and the client sends the update request only to a consistency server that holds the master of replicas. A possible extension to the protocols is to introduce a negotiation protocol for having several masters or update everywhere but do not discuss this further and leave it open for future work.

7.5 Conclusion and Future Work

The presented Consistency Service is a novel Grid service which has potentials to be used in a Data Grid for replica update synchronisation. Based on the architecture, update protocols and preliminary prototypes that have been done, we will continue this work within the DataGrid project.

An alternative to replica updates is to forbid updates on existing files but create new versions for each new file. This includes a version management and storing version numbers for physical files in the replica catalogue. Similar to the modification time, each time a replica is selected for either read or write access, the version needs to be checked in order to retrieve the most up-to-date version of a file. A default assumption can be that a query to the replica catalogue always retrieves the physical filename of a replica with the latest version number. Versioning of files has the advantage that it is very easy to track all changes on a file and going back to a previous version but this comes at the cost of higher storage requirements.

Chapter 8

Access Optimisation for Replicated Data

In all our previous chapters we discussed aspects of the replication process starting with creation of data, efficiently and securely replicating (transferring) data from one site to another, managing meta data about replicas in replica catalogues and finally updates on existing data. All this prepares the replicated data store and thus the Data Grid for end users to actually access data. We now can go into detail on how to provide optimal access to data using Grid services discussed before and we also identify additional sources of information required for a replica selection process. In detail, we elaborate on replica selection and a Replica Optimiser module for a Replica Manager (see architecture in Section 4.1). The granularity for replica selection is again at the file level.

8.1 Introduction

We first introduce the access optimisation process through a motivating example, identify the required infrastructure and then propose an architecture and a simple API for a replica selection service. Finally, existing related work is mentioned.

8.1.1 A Motivating Example

One of the main aims of data replication is to provide reduced access times to data. Rather than having a single storage location with distributed users, we have distributed and replicated data stores as well as distributed users. We motivate this by an example similar to the scenario in Section 7.1.1. We assume partially replicated data stores rather than fully replicated ones meaning that not all files are replicated everywhere. As depicted in Figure 8.1 a user at site Fermilab wants to access `file1.dbf` which is replicated and available at CERN, Italy and France but not locally at Fermilab.

At each of the four sites Grid middleware is installed and hence the user at Fermilab sends her request for `file1.dbf` to the local Replica Manager at Fermilab. It is now the task of the Replica Manager to provide the file locally at the data store (Storage Element) in Fermilab. In detail, the Replica Manager first needs to contact the replica catalogue in order to find out about the physical location of the file and then initiate the file transfer. Three replicas of the requested file exist at three different remote sites, and the Replica Manager needs to decide which is the "best" replica and thus the best location from where to get the requested file as quickly as possible. "Best" from the user's point of view means that the requested file is created locally with a minimal delay between request and actual local access to the file. In other words, the *response time* for accessing `file1.dbf` needs to be minimised. This corresponds to the

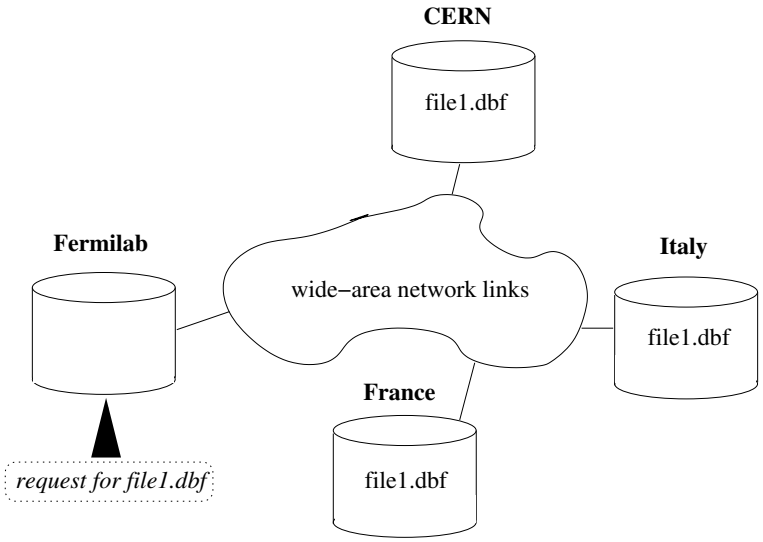


Figure 8.1: A user at Fermilab requests file1.dbf which is not available locally but at three remote sites.

minimisation of data migration as pointed out in our cost model in Section 4.2.4. Since all the three sites might have storage hardware and network connections with different access times, the replication process from CERN might take 5 minutes whereas a file transfer from France or Italy might take 8 or 10 minutes. In this case, the replica at CERN is the "best" location and a data transfer from CERN to Fermilab is initiated.

As we have identified in Chapter 4, the actual replica selection is done by the Replica Optimiser[1] (also discussed in [19] and [65]), a module used by the Replica Manager. In [65] we outlined a preliminary replica selection service but here we discuss details for efficient replica selection.

Definition: *Replica selection* is the process of selecting a single "best" replica from a set of identical replicas where best means that the response time for accessing a file locally is minimal. If replicas are only available at remote sites, the access time also includes the data transfer time from the remote to the local site.

8.1.2 Infrastructure and Architecture

Replica selection is driven by a set of static and dynamic influences in a Data Grid as pointed out in our cost model in Section 4.2.3. Whereas static parameters like available network bandwidth or size of a file to be transferred are rather easy to obtain, dynamic influences like current network throughput and data server load need more complex performance measurement tools and will be further discussed in Section 8.2.

Consequently, the basic architecture of a replica selection service implemented in a Replica Manager looks like follows (see also Figure 8.2): The Replica Optimiser module gets replica information from the replica catalogue service and then requires static and dynamic information from the respective information services. If dynamic information is analysed, it is important for a good replica selection service to give rough predictions about near future data transfer rates. Thus, an information service providing dynamic information has to keep internal statistics and histories about past performance measurements and should have a basic functionality for

[1]The Replica Optimiser as defined in [65] has a much broader scope including to aid load balancing etc. Replica selection with time estimation is just a subset we are discussing in more detail here.

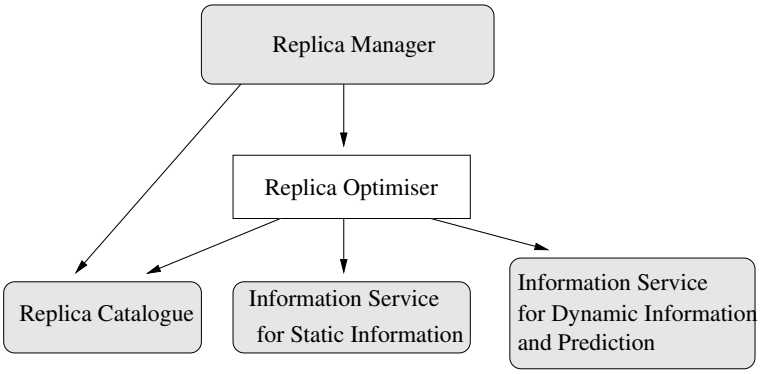


Figure 8.2: Architectural components of a replica selection service.

predicting performance parameters. This is important for predicting how long it might take to transfer a file from one site to another taking into account the current and the possible future network throughput.

A simple API to the Replica Manager for replica access optimisation looks like follows and is assumed for further discussions in this chapter:

```
pfn = getBestPhysicalFileName(LFN lfn)
```

A request for a file is done via its logical filename and a physical filename is returned by the replication service, i.e. the Replica Manager. `lfn` and `pfn` are string values that satisfy the definitions from Section 4.1.2. As for our previous example, `lfn` could be `file1.dbf` which is passed to the Replica Manager and then to the Replica Optimiser module. The latter looks up the list of all possible physical filenames in the replica catalogue, retrieves replica locations and then tries to get all possible performance information about point-to-point connections Fermilab-CERN, Fermilab-Italy and Fermilab-France. The site with the minimal response time is identified and the physical filename for that site is returned which may be `host1.cern.ch/data/file1.dbf` for our example.

8.1.3 Related Work

In current database technology, replica selection is not yet implemented and used. Although the theory behind replica selection based on performance monitoring and prediction is rather straight forward, practical performance measurement and prediction are not trivial. For instance in Objectivity (see Section 2.6.1) no performance parameters are used for the replica selection process. Replica selection is a very new field in Grid research but has longer traditions in the Internet community. Here, we briefly state some relevant related work.

In Grid research, replica selection is first mentioned in [126] with respect to the Globus toolkit. The replica selection architecture is similar to our proposed architecture and also builds on existing Grid services. Main emphasis is put on a storage broker where replica selection is done in three phases: a search phase (information services are queried), a match phase (a file request is matched to Condor class adds) and an access phase (the actual file transfer). The basic functionality of the broker is discussed but dynamic performance information (network bandwidth, server loads, etc.) is currently not used for the replica selection process. Further related work on the Earth Science Grid can be found in [5].

8.2 Performance Parameters

As we pointed out, the response time for replica access is the time it takes to open a file locally for read access. If the file is not available locally, it first needs to be transferred via a file transfer service from a remote site to the local destination and then a file handle can be passed to the local application. The transfer speed mainly depends on the network bandwidth[2] (BW_{WAN}), the current network throughput (f_{WAN}) as well as the capability of the file transfer server and the remote disk subsystem ($speed_{data_server}$). We categorise these influences as performance parameters for network and data servers and discuss them here in detail. To sum up, a performance parameter analysis is required to determine latencies in the file transfer.

8.2.1 Network

The simplest way to determine the latency in a wide-area network connection is to use the static network bandwidth. A very simple replica selection process needs to store information about all possible end-to-end network links and then compare their latencies. Another method is to use the round trip time (RTT)[3] of a single message. If two links have the same bandwidth, a link with lower RTT is preferable.

Much more important than static parameters is the dynamically changing, current throughput of a network link. This requires particular monitoring tools that permanently monitor the traffic between two sites. The traffic history needs to be logged and some guess about the future transfer speed can be made. Note that retrieving the current throughput of a network link depends on several network parameters like the TCP buffer size, number of parallel streams etc. that all have to be taken into account and may be error prone. Moreover, performance prediction might give incorrect answers as well but is a good starting point for further research and needs to be tuned to the particular network link. Finally, just the availability of a network link needs to be included in a replica selection process.

There exist several tools for monitoring different kinds of network parameters: Iperf [37] (maximum TCP or UDP bandwidth), pipechar [69] (for probing network bottlenecks), NIMI [1] (application wrapper for network monitoring) and Network Weather Service (NWS) [134](does such measurements and network performance predictions).

8.2.2 Data Server

Not only the network connection contributes to the transfer latency but also the *load of the data server* and its capability to serve multiple requests in parallel. We do not discuss the I/O bottleneck problem of data servers nor their disks but discuss how the influence of the data server can be identified.

The file copier service introduced in Chapter 4 can be a specialised FTP server like GridFTP and we now assume such a server for further discussion. Thus, the FTP server is the primary data server for wide-area access but data can still be accessed locally through conventional file system servers. Hence, we have to distinguish between load on the FTP server and the local file system server.

A possible way to keep track of traffic of an FTP server is to log all outgoing and incoming data transfer. In detail, also the file copier service can trigger the monitor process which records all data transfer in bytes, the average throughput, the number of parallel streams, buffer sizes etc. Based on this history a simple guess can be made about the possible load of the FTP server. In addition, the used CPU time of the FTP daemon can be monitored. If the

[2] minimum of network card, wide-area link and other possible network bottlenecks of routers, switches etc.

[3] The RTT alone may not be a good metric for an asymmetric data path and one-way delay may be more appropriate than RTT.

FTP server is contacted directly, more detailed performance monitoring is possible: number of concurrent connections, data channel traffic, control channel traffic etc.

Whereas FTP server load is rather easy to measure, file servers in a file system are more difficult to monitor and require specialised APIs which can be provided for Grid tools like GridFTP etc. In the cost model in Section 4.2.3 we pointed out that the number of jobs waiting to be served by a data server can further be used as a performance parameter. Thus, a *data server queue* needs to contain the number of requests to be served including the required file size for each request. Since both, FTP server and file server, normally do not have easy accessible external user interfaces to such a queue, a specialised queue service for a disk or a disk pool can take over this task. In particular, a disk pool manager can serve such requests. On the contrary, an FTP server can be configured in such a way that every request for a file which cannot be served immediately, is put in a request queue accessible via an external interface.

If hierarchical storage or mass storage systems are used, additional latency occurs for data transfer. In particular, a requested file might be stored on a tape system and first needs to be staged from tape to disk before it can be transferred to remote sites. We define this as a general mass storage system (MSS) latency parameter $latency_{MSS}$. For a MSS there is normally an internal request queue available that can be included in the replica selection process. As regards query optimisation for files replicated to several mass storage systems, we provide more detailed discussions in [111].

8.3 Replica Selection

Having identified possible performance parameters that influence the replica selection process, we now present a cost model that takes the above parameters into account. The Replica Optimiser can then use this cost model for deciding what is the "best" replica for a given site. Since replica selection also needs to consider inconsistent replicas, we discuss the impact on this, too. Finally, we discuss further replica selection issues.

8.3.1 A Cost Model for Replica Selection

Selecting a "best" replica from a set of potential identical replicas can be regarded as an optimisation problem where we want to find a minimum for the response time to data access. We present a cost model for such an optimisation problem that takes into account requests for single files and then we elaborate how a request for several files can be handled. In detail, we want to minimise the transfer time of a file from one storage location to another. Using the replica selection API defined in Section 8.1.2, several point-to-point connections between the local site ($site_{local}$) and remote sites ($site_{remote}$) where physical files are located have to be considered. The network link with the minimal file transfer time needs to be identified and represents the minimum in the optimisation problem where i denotes the number of remote sites where a replica exists:

$$\mathbf{min}(file_transfer_i(site_{local}, site_{remote}i)) \tag{8.1}$$

We now need to define functions to determine the transfer speed $file_transfer_i$ for single links. Based on the amount of available performance parameters, several functions are possible. The functions need to return a value indication the time to transfer the file from $site_{remote}i$ to $site_{local}$. Note that the returned value is not expressed in seconds but rather considered to be a value to compare different network links. The lower the value, the "better" the network link[4].

[4]Note that with "network link" we also include data server speed.

We present a set of functions with varying input parameters and thus also different levels of usefulness. We assume a file of size $size_{file}$ to be transferred. A general metric for the file transfer depends on the network latency and additional latency of a data server:

$$file_transfer_i(site_{local}, site_{remote}i) = latency_{network} + latency_{data_server} \tag{8.2}$$

The network latency ($latency_{network}$) is determined like follows:

- $\frac{size_{file}}{BW_{WAN}}$: this is the simplest function considering only static information and is not very accurate
- $\frac{size_{file}}{BW_{WAN}} * RTT$: The delay of a network link (RTT expressed in ms) makes the comparison more accurate and considers physical distance between two sites.

More accurate functions use current network throughput (f_{WAN}) instead of maximal possible bandwidth (BW_{WAN}).

- $\frac{size_{file}}{f_{WAN}}$ or: $\frac{size_{file}}{f_{WAN}} * RTT$

The question that arises now is how to determine the current throughput f_{WAN} and how long is it valid. Possible methods are to use the history of measured performance of a network link and use aggregation functions like mean or average. How much past information is required to give a good guess, is not obvious and requires detailed performance measurement and tuning. We leave this open as a networking research issue. Network performance prediction tools also predict the current estimated throughput. Thus, the parameter f_{WAN} can be supplied by such a tool.

We now discuss data server latency ($latency_{data_server}$) which seems to be more difficult to determine than the network latency and thus might not be available for some sites. In this case, the $latency_{data_server}$ equals to 0 and does not influence the cost model at all. If the server latency is available, a time delay in seconds is given. In detail, the data server latency consists of the following three parameters which all depend on the file size:

$$latency_{data_server} = server_load + request_queue + latency_{MSS} \tag{8.3}$$

$latency_{MSS}$ is only added if a Mass Storage System is used. The latency factor can be the average time for serving a file and is gained by monitoring the staging behaviour of the specific Mass Storage System. *server_load* as well as *request_queue* cannot be retrieved directly and need some additional transformation function provided by the data server software. Server load can be expressed as a time delay for serving an additional file request. The same is true for the request queue.

A request for a single file can be estimated relatively easy compared to a request for a set of files. Whereas in the first case the time of the estimation and the actual transfer is small or even 0, for a large amount of files the latency parameters might not be useful anymore and need to be re-calculated after some time.

Example

We illustrate the discussion by an example using our introductory Data Grid with four sites and a file request for `file1.dbf` is done at Fermilab. The file size is 2 GB. Table 8.1 lists static and dynamic performance information for end-to-end links from Fermilab to CERN, Italy and France.

Since we have values for all our possible performance parameters, we insert the values into the equations 8.2 and 8.3 and get the following transfer rates:

Fermilab	CERN	France	Italy
BW_{WAN}	45 Mbps	45 Mbps	45 Mbps
f_{WAN}	3 MB/s	3 MB/s	2 MB/s
RTT	0.11 s	0.12 s	0.115 s
$server_load$	1 s	3 s	10 s
$queue_size$	0 s	0 s	1 s
$latency_{MSS}$	10 s	40 s	50 s

Table 8.1: Performance parameter for a replica selection service

```
file_transfer1(Fermilab, CERN)  :  50.1
file_transfer2(Fermilab, France):  85.7
file_transfer3(Fermilab, Italy) : 101.9
```

The minimum transfer rate is over the link Fermilab-CERN: this is the expected output of the Replica Optimiser by minimising equation 8.1.

Note that the the server latency has a rather high impact on the cost function due to access times in seconds. For longer latency times, the parameters might be adjusted by a factor in the range between $0 < x \leq 1$.

8.3.2 Replica Selection and Data Consistency

Replica section also needs to deal with files that are updateable. Since we assume an asychnronous update replication process (see Chapter 7) with certain delays and temporary inconsistencies of replicas, a replica that promises best transfer time might not be up-to-date and end-users might want to be informed about consistency issues. In the replica catalogue a physical file that is updateable has to be marked as such and a selection process of the Replica Optimiser needs to compare file modification times. A default setting can be that only the latest version(s) of replicas (having the most up-to-date modification time) is/are considered for the replica selection process.

8.3.3 Discussion

In the configuration of the replica selection service the treatment of local replicas has to be configurable. If a file is transferred from a remote site to the local one, it can be *cached locally* as long as it is required by the local application but not made public through a replica catalogue entry. In this case, the new file location is not known to Grid users but just to the local application. On the other hand, a file can also be integrated into the replica catalogue and a new replica is created locally.

Up to now we assumed that access to data is done directly on storage systems and CPU resources are connected to the data stores (Storage Elements). In the DataGrid environment files often cannot be accessed directly at the data store but first need to be transferred to local disks of computing farms. The above cost model then needs to be extended by another latency factor $latency_{copy_locally}$ which creates a copy of the requested file at the computing farm within the assumed local-area network.

8.4 Advanced Issues

Replica selection is not the only access optimisation strategy in a Data Grid. Since the use of Data Grids and the access patters have in important impact on the access to replicated files,

we point out some implications for Grid application.

8.4.1 Implications for Grid Applications

We claim that the usage of files that are replicated in a Data Grid will have implications for Grid applications that are new to the High Energy Physics user community [117]. We address explicitly some possibilities of how a replicated file instance can be accessed and give some implications of read ahead assumptions. Since Objectivity/DB is currently in use at different sites, we discuss some performance issues for accessing database files in a Grid environment.

Reading data ahead (pre-fetching) before using them is a commonly used optimisation technique. However, this imposes a problem, which we illustrate by an example. Let us assume a file size of 2 GB and a program that wants to access 5 objects in the file where each object has a size of 10 KB. It is obvious that we want to transfer the requested objects rather than the whole file, and having unnecessary information transferred over the network. There is a clear need for a mapping instance that maps requested objects to files in order to determine in which files the requested objects reside. Based on the type (class definition in the data definition language of the ODBMS) of the objects it can be roughly determined how much data needs to be transferred over the network (depending on how much dynamically allocated information is stored in the object). This problem cannot be solved when only the "open" file operator is used. Only at runtime the application can determine which data is needed and a pre-fetching of requested objects can tackle this problem. We can summarise this as a query optimisation problem where access to the data shall be optimised by providing only the necessary objects that are required. Related work can be found in [123, 12].

Pre-fetching can also be addressed at the file level. We assume that a user is aware that files are distributed to different sites in the Grid and the time to access data very much depends on the location of the requested file. We further assume that an object in a file can only be accessed when the whole file is available locally at the client site. This is assumed for simplicity and we neglect the fact of accessing some parts of a file remotely. In the worst case, all the requested files are available remotely and need to be transferred to the local site. If the size of a file is large and also the amount of requested files is high, the I/O part of an application takes a long time before the actual computation on the data can be done. The application can give a hint to the system to tell it how long it would need to serve the required files and when the computation can be started. Thus, data can be pre-fetched before the application is using the data. In the Grid environment, the necessary information for pre-fetching files can be provided by diverse monitoring tools and information services.

This is also a *sociological aspect* of Grid applications in the sense that a user of a data-intensive application requests its data in advance. One can also reserve the necessary network bandwidth and start the the application program at a certain point in the future when the network throughput is available.

8.4.2 Dynamic Replication

In 8.3.3 we outlined that files can either be cached temporarily or replicas can be created. An advanced feature of a replicated system is that replicas get created and deleted dynamically as access patterns change. We see this as a field of future research within the DataGrid project and now only point out some related work in this field.

Recent work on dynamic replication in a Data Grid with emphasis on read-only files can be found in [100]. Several replication strategies and simulation results are presented. Dynamic service replication is discussed in [77] and builds on the Legion system. Cost models for replica selection are based on network and server latencies and consider similar aspects like in our cost model.

In [133] adaptive data replication algorithms are proposed that change the scheme of an object (i.e. the set of processors at which the object is replicated) as changes occur in the read write pattern of the object. This is in opposition to a static replication schema. Furthermore, a protocol called "Primary Missing Writes" is proposed that can be combined with dynamic replication.

Dynamic replica service creation is discussed in [86] and called "fluid replication". Clients interact with a service and monitor the observed performance. If the performance becomes poor because of increased network load or client mobility, a new replica is created. Replication services can only be created on a "Way Station", i.e. a dedicated node. A client is responsible to create a new replica and then sends all requests to the new service replica. A file server is a possible application.

8.5 Conclusion and Future Work

We have discussed in detail several replica selection issues and a cost model for the actual selection process that needs to be performed by the Replica Optimiser module. We only discussed preliminary replica selection and access optimisation issues for files but we plan more advanced query estimation and creation of query execution plans in the DataGrid project. Since also replica selection based on volatile and dynamic performance information is hardly used in a production system today, this field is open for further research.

Chapter 9

Conclusion

Sometimes the word "Grid" is used as hype and some people regard it as the "second Internet". The former might well be the case but the latter is not really true since Grid technology builds on Internet technology. So what is then really left and new in the Grid? If we regard the Grid as a special purpose, private network of computers or local-area networks connected via wide-area network links, a Grid brings a kind of homogeneous view on the distributed computing and storage resources. In the particular case of replication, there is a definite and clear requirement of dealing efficiently with file transfers and replica management. Features are similar as in a distributed database management system but there are new challenges to face: large amounts of data and large, world-wide distributed user groups.

Although Data Grids are a rather new research area, the research community is growing more and more; not alone through sample applications like CERN's world-wide distributed computing model. Whereas job scheduling and resource management are already rather established sub domains in Grid computing [74], data management still has not been addressed sufficiently to satisfy the stringent needs of the scientific community to manage replicated data on several sites in the Grid. In this thesis we proposed several architectural solutions and showed through prototype and production quality software that Data Grids are reality already today and not just a vision for tomorrow.

Replication of (database) files requires a particular software infrastructure including Grid middleware for security and file transfer. Besides architectural aspects for data management in the context of the European DataGrid project, we proposed a data replication architecture and a corresponding implementation called Grid Data Mirroring Package (GDMP). This software is in production use in the CMS experiment and is also deployed in DataGrid's first testbed in autumn 2001. In addition to the core replication system for read-only files, we proposed a distributed file replica catalogue and a corresponding implementation. We covered a large spectrum of file replication issues by including considerations about replica synchronisation (updates to replicated files) and replica access optimisation. Consequently, this thesis also serves as a guide for dealing with several aspects of file replication in a Data Grid environment.

Having contributed to the initial proposal of the European DataGrid, being active in first prototype implementations and production deployment of the software, we will continue our work in the DataGrid project and further contribute to the Grid research community.

Acknowledgements

I finish my thesis with small anecdotes in my acknowledgement section. Many friends and colleagues have contributed to my success, and I want to thank them all here.

I started my studies in Vienna, a wonderful city in Austria, and begin the journey of my Ph.D. period there. Going from Vienna to CERN (Geneva, Switzerland) has been a challenging and interesting undertaking from the very beginning. I still remember the time when my Ph.D. supervisor *Erich Schikuta* spoke about the possibility to do a Ph.D. at CERN, famous for the Web. Erich is the first person I want to thank for the great opportunity he proposed to me. This opened the door to a new, challenging research environment. I want to thank also my second Ph.D. supervisor in Vienna and currently in Australia *Gerald Quirchmayr*.

Before I actually "go" to Geneva, I include my friend and colleague *Stepan Havel*. He might not even remember anymore but Stepan really created my interest in object-oriented databases - one of the driving factors for my decision to go to CERN. Thank you to *Wolfgang Hoschek*, also a friend and colleague in the DataGrid project, who was my very first contact person at CERN and helped me to overcome the first obstacles.

Now let us go from Vienna to Geneva and CERN. The person whom I owe the greatest thank during my stay at CERN is *Ian Willers*, my Ph.D. supervisor at CERN. He has always been like a father and treated his Ph.D. students (Koen Holtman, Frank van Lingen and me) like his "boys". He had always time for my concerns, wishes and ideas and was very encouraging in all I did. Since I believe that family is the most important thing in life, I want to thank *my parents* who were always there for me during all areas of my life. Life becomes even easier if there is a second person around that goes the same way like you. This second person is my replica `:-)` - yes, I shall call him like that. I always enjoy this joke and it fits in the context of my thesis. Sure, this person is *my twin brother Kurt*.

Now again back to the CERN family. Already at the CERN School of Computing in Poland, September 1999, one of the lecturers said that the HEP community is like a big family. Thank you to *Koen Holtman*, my friend and colleague in the CMS computing group and later at Caltech. He was a great technical help and also my first publication at CERN was a mutual work with Koen.

Koen left for Caltech and another person from Caltech arrived at CERN: *Asad Samar*. Originally, we were working on some Perl scripts for the CMS production system but later we initiated, designed and developed the GDMP project. Asad has been a great friend and colleague and we were successful in developing our Grid tool. Thanks also to *Tony Wildish* for his discussions and input for GDMP.

GDMP is the main software product that resulted from the practical work of this thesis and helped me to establish a personal success in the Grid research community. I mentioned already in the introduction that Asad Samar was my "co-pilot" in this software pilot project and he deserves my greatest thank and appreciation. The project grew and became more important and so we extended our GDMP team. "Shukria" to *Shahzad Muzaffar* alias "Ali" from Fermi National Laboratory near Chicago for his invaluable contribution in mainly coding and some design discussions. "Grazie" to *Flavia Donno* (INFN Pisa) for her contribution on configuration

management, packaging and lots of discussions on systems setup and configuration. Thanks to *Andrea Domenici* (INFN Pisa).

"The Grid": this was indeed the driving force in my Ph.D. period. When I arrived at CERN and did my first prototypes in replication, "nobody" knew about the Grid (including myself). in February 2000 I joined the first meetings here at CERN and was impressed and motivated from the beginning and joined the European DataGrid project (initiated by CERN). Thank you to my group leader *Werner Jank* and the CMS computing project leader *Martti Pimiä* for their support during my Ph.D. period. My excitement for the Grid was also inspired by the "Grid Gods" (as I always call them) *Ian Foster* and *Carl Kesselman*. Thank you to both of you for fruitful discussions and fun at various meetings and conferences. Thanks also to *Rajkumar Buyya* for useful comments.

Thank you to *Ben Segal*, the work package manager for Data Management in the DataGrid project. Although I will continue the work within this work package, I want to thank my colleagues already now for all the good and interesting work we did together. Thanks to *Brian Tierney* (also for all your cool BBQs, parties, etc.), *Peter Kunszt* (for your constructive comments), *Dirk Düllmann* (for help and discussions on Objectivity, replication etc.) and *Javier Jaen Martinez*.

I spent almost three entire months in the Silicon Valley of sunny California where I worked with *Andy Hanushevsky* at SLAC (Stanford Linear Accelerator Center, Stanford University) on replica catalogue issues. Thanks for the interesting work we did and the possibility to visit SLAC.

Last but not least I want thank some friends that helped me to have a good time here in Switzerland and France. *Veronique Lefebure* is not only a great help to improve my French but also a good friend (including her baby "littol" Jerome (before he was born I gave him the name "Mister X") and her husband Olivier). I also do not want to miss my Israel dance classes with *Ilse* and *Lennart Jirden*.

Danke !
Heinz Stockinger

Bibliography

[1] Andrew Adams, Tian Bu, Timur Friedman, Joseph Horowitz, Don Towsley, Ramon Caceres, Nick Duffield, Francesco Lo Presti, Sue. B. Moon, Vern Paxson. The Use of End-to-End Multicast Measurements for Characterizing Internal Network Behavior, *IEEE Communications*, May 2000.

[2] Divyakant Agrawal, Amr El Abbadi, R. Steinke: Epidemic Algorithms in Replicated Databases, *Proc. 16 ACM Sigact/Sigmod Symposium on the Principles of Database Systems*, Tucson, Arizona, 1997.

[3] Divyakant Agrawal, Gustavo Alonso, Amr El Abbadi, Iona Stanoi. Exploiting Atomic Broadcast in Replicated Databases, *Proceedings of Euro-Par'97*. Passau, Germany, August 1997.

[4] Bill Allcock, Joe Bester, John Bresnahan, Ann Chervenak, Ian Foster, Carl Kesselman, Samuel Meder, Veronica Nefedova, Darcy Quesnel, Steve Tuecke. Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing, *18th IEEE Symposium on Mass Storage Systems and 9th NASA Goddard Conference on Mass Storage Systems and Technologies*, San Diego, April 17-20, 2001.

[5] Bill Allcock, Ian Foster, Veronika Nefedova, Ann Chervenak, Ewa Deelman, Carl Kesselman, Jason Lee, Alex Sim, Arie Shoshani, Bob Drach, Dean Williams. High-Performance Remote Access to Climate Simulation Data: A Challenge Problem for Data Grid Technologies, *Supercomputing 2001*, Denver, Texas, November 2001.

[6] Bill Allcock, Ann Chervenak, Ian Foster, Carl Kesselman, Chuck Salisbury, Steve Tuecke. The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets, *Journal of Network and Computer Applications: Special Issue on Network-Based Storage Services*, vol. 23, no. 3, p.187-200, July 2000.

[7] Todd Anderson, Yuri Breitbart, Henry Korth, Avishai Wool. Replication, Consistency, and Practicality: Are These Mutually Exclusive?, *ACM SIGMOD International Conference on Management of Data*, Seattle, Washington, June 2-4, 1998.

[8] Olof Barring, Jean-Philippe Baud, Jean-Damien Durand. CASTOR Project Status, *Computing in High Energy Physics (CHEP 2000)*, Padova, February 2000.

[9] C. Baru, Reagan Moore, A. Rajasekar, M. Wan. The SDSC Storage Resource Broker, *CASCON'98*, Toronto, Canada, Nov. 30 - Dec.3, 1998.

[10] Jim Basney, Miron Livny, and Todd Tannenbaum, High Throughput Computing with Condor, *HPCU news*, Volume 1(2), June 1997.

[11] Micah Beck, Terrry Moore. The Internet2 Distributed Storage Infrastructure (I2-DSI) Project, 1999.

[12] Luis M. Bernardo, Arie Shoshani, Alex Sim, Henrik Nordberg. Access Coordination of Tertiary Storage for High Energy Physics Applications. *IEEE Symposium on Mass Storage Systems*, College Park, MD, USA, March 2000.

[13] Nassem Bhatti, Jean-Marie Le Goff, Waseem Hassan, Zsolt Kovacs, Richard Peter Martin, McClatchey, Heinz Stockinger, Ian Willers. Object Serialisation and Deserialisation Using XML, *10th International Conference on Management of Data (COMAD 2000)*, Pune, India, 14 -16 December 2000.

[14] Peter Braam, The Coda Distributed File System, *Linux Journal*, Number 50, June 1998.

[15] Peter Braam, Michael Callahan, Phil Schwan. The InterMezzo Filesystem, *O'Reilly Perl Conference*, Monterey, California, August 21-24, 1999.

[16] Yuri Breitbart, Henry Korth. Replication and Consistency: Being Lazy Helps Sometimes, *Proc. 16 ACM Sigact/Sigmod Symposium on the Principles of Database Systems*, Tucson, Arizona, 1997.

[17] Yuri Breitbarth, Henry Korth. Data Replication Gaining Popularity, *IEEE Concurrency*, June 1999.

[18] Yuri Breitbart, Raghavan Komondoor, Rajeev Rastogi, S. Seshadri, Abraham Silberschatz. Update Propagation Protocols For Replicated Databases. *ACM SIGMOD International Conference on Management of Data*, Philadelphia, Pennsylvania, June 1-3, 1999.

[19] Paolo Busetta, Mark Carman, Luciano Serafini, Kurt Stockinger, Floriano Zini. Grid Query Optimisation in the Data Grid Query Optimisation Task - WP2, European DataGRID Project, *IRST Technical Report 0109-01*, Istituto Trentino di Cultura, September 2001.

[20] Rajkumar Buyya, Heinz Stockinger, David Abramson, Jonathan Giddy. Economic Models for Regulating Resource Allocation and Brokering in the Global Grid Computing Environments, *Commercial Applications for High-Performance Computing, SPIE's International Symposium on The Convergence of Information Technologies and Communications (ITCom 2001)*, Denver, Colorado, August 20-24, 2001.

[21] Stefano Ceri, Guseppe Pelagatti. Distributed Databases - Principles and Systems, McGraw-Hill Computer Science Series, 1985.

[22] Stefano Ceri, Maurice A. W. Houtsma, Arthur M. Keller, Pierangela Samarati. Independent Updates and Incremental Agreement in Replicated Databases, *Distributed and Parallel Databases, Volume 3*, Kluwer 1999.

[23] Ing-Ray Chen and Ding-Chau Wang. Analysis of Replicated Data With Repair Dependency, *The Computer Journal*, Vol. 39, No. 9, 1996.

[24] Ing-Ray Chen, Ding-Chau Wang and Chih-Ping. Chu. User-Perceived Availability and Response-time in Voting-based Replicated Systems: A Case Study, *2nd IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET'99)*, Richardson, Texas, March 1999.

[25] David Choy, Patricia Selinger. A Distributed Catalog for Heterogeneous Distributed Database Resources. *International Conference on Parallel and Distributed Information Systems (PDIS 1991)*, Miami Beach, Florida, December 4-6, 1991.

[26] Parvathi Chundi, Daniel J. Rosenkrantz, S. S. Ravi. Deferred Update Protocols and Data placement in Distributed Databases, *International Conference on Data Engineering*, IEEE Computer Society, New Orleans, Louisiana, March 1, 1996.

[27] CMS Computing Technical Proposal, CERN/LHCC 96-45, December 1996.

[28] George Coulouris, Jean Dollimore, Tim Kindberg. Distributed Systems: Concepts and Design, 2nd Edition, Addison-Wesley 1998.

[29] Bert Dempsey, Debra Weiss. On the Performance and Scalability of a Data Mirroring Approach for I2-DSI, *Network Storage Symposium (NetStore 1999)*, Seattle, WA, October 14 -15 , 1999.

[30] Distributed Computing Environment (DCE), http://www.opengroup.org/dce/

[31] Werner Dreyer, Klaus Dittrich. A Meta Model and an Infrastructure for the Non-Transparent Replication of Object Databases, *ACM CIKM International Conference on Information and Knowledge Management*, McLean, Virginia, November 6-11, 2000.

[32] Dirk Düllmann, Wolfgang Hoschek, Javier Jaen-Martinez, Asad Samar, Ben Segal, Heinz Stockinger, Kurt Stockinger. Models for Replica Synchronisation and Consistency in a Data Grid. *10th IEEE International Symposium on High Performance and Distributed Computing (HPDC-10)*, San Francisco, California, August 7-9, 2001.

[33] European DataGrid Project, http://www.eu-datagrid.org

[34] Andre Eickler, Alfons Gerlhof, Donald Kossmann. A Performance Evaluation of OID Mapping Techniques, *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Zurich, Switzerland, September 1995.

[35] Todd Ekenstam, Peter Reiher, Charles Matheny, and Gerald Popek, "The Bengal Database Replication System, *The Journal of Distributed and Parallel Databases*, Volume 9, Number 3, May 2001.

[36] Enstore: http://www-isd.fnal.gov/enstore/design.html

[37] Jim Ferguson, Ajay Tirumala, Iperf version 1.2, http://dast.nlanr.net/Projects/Iperf/, May 2001.

[38] Roy Fielding, James Gettys, Jeffrey Mogul, Henrik Nielson, Larry Masinter, Paul Leach, Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, Request for Comments (RFC) 2616, June 1999.

[39] Ian Foster, Carl Kesselman (editors), The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann Publishers, USA, 1999.

[40] Ian Foster, Alain Roy, Volker Sander. A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation, *8th International Workshop on Quality of Service*, Pittsburgh, June 5-7, 2000.

[41] Ian Foster, Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit, *International Journal of Supercomputer Applications*, 11(2):115-128, 1997.

[42] Ian Foster, Carl Kesselman, Gene Tsudik, Steve Tuecke. A Security Architecture for Computational Grids, *5th ACM conference on Computer and Communications Security*, San Francisco, California, November 2 - 5, 1998.

[43] Rainer Gallersdörfer, Matthias Nicola. Improving the Performance in Replicated Databases Through Relaxed Coherency. *21th International Conference on Very Large Data Bases (VLDB)*, Zurich, Switzerland, September 11-15, 1995.

[44] Rainer Gallersdörfer, Matthias Jarke, Matthias Nicola. The ADR Replication Manager, *International Journal of Cooperative Information Systems (IJCS)*, Vol. 8, 1999.

[45] German Gancio, Steve Fischer, Tim Folkes, Francesco Giacomini, Wolfgang Hoschek, Dave Kelsey, Brian Tierney. The DataGrid Architecture, Geneva, Switzerland, July 2001.

[46] Garth Gibson, David Nagle, Khalil Amiri, Fay Chang, Eugene Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, Jim Zelenka. File Server Scaling with Network-Attached Secure Disks. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Seattle, Washington, June 15-18, 1997.

[47] D.K. Gifford. Weighted Voting for Replicated Data. *ACM-SIGOPS Symp. on Operating Systems Principles*, Pacific Grove, December 1979.

[48] Global Grid Forum, http://www.gridforum.org

[49] Jim Gray, Pat Helland, Patrick O'Neil, Dennis Shasha. The Dangers of Replication and a Solution. *ACM SIGMOD International Conference on Management of Data*, Montreal, Quebec, Canada, June 4-6, 1996.

[50] Grid Physics Network (GriPhyN): http://www.griphyn.org

[51] Andrew Grimshaw Wm Wuld and the Leagion Team. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, Volume 40, Number 1, January 1997.

[52] Mehnaz Hafeez, Asad Samar, Heinz Stockinger. A Data Grid Prototype for Distributed Data Production in CMS, *VII International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT2000)*, Chicago, Illinois, October 2000.

[53] Andrew Hanushevsky, Marcin Nowak. Pursuit of a Scalable High Performance Multi-Petabyte Database. *IEEE Symposium on Mass Storage Systems*, March 15-18, 1999.

[54] Andrew Hanushevsky. Objectivity/DB Advanced Multi-threaded Server (AMS) www.slac.stanford.edu/∼abh/objy.html, April 2000.

[55] Sato Hiroyuki. Objectivity AMS on the Wide Area Network, presentation at the RD45 Workshop, http://wwwinfo.cern.ch/asd/rd45/workshops/jan-feb2000/index.html, Febr. 2000.

[56] JoAnne Holliday, Divyakant Agrawal, Amr El Abbadi. The Performance of Database Replication with Group Multicast, *IEEE International Symposium on Fault Tolerant Computing (FTCS29)*, Madison, Wisconsin, June 15-18, 1999.

[57] JoAnne Holliday, Robert Steinke, Divyakant Agrawal, Amr El Abbadi. Epidemic Quorums for Managing Replicated Data. *19th IEEE International Performance, Computing, and Communications Conference (IPCCC 2000)*, Phoenix, Arizona, February 2000.

[58] Koen Holtman, Peter van der Stok, Ian Willers. Automatic Reclustering of Objects in Very Large Databases for High Energy Physics, *Proc. of IDEAS '98*, IEEE Computer Society, Cardiff, UK, 1998.

[59] Koen Holtman, Peter van der Stok, Ian Willers. A Cache Filtering Optimisation for Queries to Massive Datasets on Tertiary Storage,*Proc. of DOLAP'99*, Kansas City, USA, November 6, 1999.

[60] Koen Holtman, Heinz Stockinger. Building a Large Location Table to Find Replicas of Physics Objects, *International Conference on Computing in High Energy Physics (CHEP 2000)*, Padova, Italy, Febr. 2000.

[61] Koen Holtman. Prototyping of CMS Storage Management, Ph.D. thesis (proefontwerp), Eindhoven University of Technology, ISBN 90-386-0771-7, May 2000.

[62] Koen Holtman. Object Level Physics Data Replication in the Grid. *VII International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT2000)*, Chicago, Illinois, October 2000.

[63] Koen Holtman, Heinz Stockinger. Building a Large Location Table to Find Replicas of Physics Objects, *Computer Physics Communications*, 2001.

[64] Wolfgang Hoschek, Javier Jean-Martinez, Asad Samar, Heinz Stockinger, Kurt Stockinger. Data Management in an International Data Grid Project. *1st IEEE/ACM International Workshop on Grid Computing (Grid'2000)*. Bangalore, India, Dec 17-20, 2000.

[65] Wolfgang Hoschek, Javier Jean-Martinez, Peter Kunszt, Ben Segal, Heinz Stockinger, Kurt Stockinger, Brian Tierney. Data Management (WP2) Architecture Report - Design, Requirements and Valuation Criteria, *DataGrid-02-D2.2-0103-1_2*, http://grid-data-management.web.cern.ch/grid-data-management/docs/DataGrid-02-D2.2-0103-1_2.pdf, Geneva, Sept 19, 2001.

[66] John Howard. An Overview of the Andrew File System. Proceedings of the *USENIX Winter Technical Conference*, Dallas, TX, Feb. 1988.

[67] HPSS: High Performance Storage System, http://hpcf.nersc.gov/storage/hpss/

[68] Richard Hurly, Soon Aun Yeap. File Migration and File Replication: A Symbolic Relationship, IEEE Transactions on Parallel and Distributed Computing, Volume 7, Number 6, June 1996.

[69] Guojun Jin, George Yang, Brian Crowley, Deborah Agarwal. Network Characterization Service(NCS). *10th IEEE International Symposium on High Performance and Distributed Computing (HPDC-10)*, San Francisco, California, August 7-9, 2001.

[70] Peter Keleher. Decentralized Replicated-Object Protocols, *18th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, April 1999.

[71] Bettina Kemme, Fernando Pedone, Gustavo Alonso, Andr Schiper. Processing Transactions over Optimistic Atomic Broadcast Protocols.

[72] Bettina Kemme, Gustavo Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement Database Replication, *26th International Conference on Very Large Databases (VLDB)*, Cairo, Egypt, September 2000.

[73] Henry F. Korth, Abraham Silberschatz, Database System Concepts, McGraw-Hill Computer Science Series, 1991.

[74] Klaus Krauter, Rajkumar Buyya, and Muthucumaru Maheswaran. A Taxonomy and Survey of Grid Resource Management Systems, *Software: Practice and Experience (SPE) Journal*, Wiley Press, to appear.

[75] Akhil Kumar, Noha Adly. HPP: A Reliable Causal Broadcast Protocol for Large-Scale Replication in Wide Area Networks, *The Computer Journal*, Volume 41, Number 2, 1998.

[76] Alla Lake, Jonathan Crawford, Raymond Simanowith, Bradley Koenig. Fault Tolerant Design in the Earth Observing System Archive. *7th IEEE Symposium on Mass Storage Systems and Eigth Goddard Conference on Mass Storage Systems and Technologies*, College Park, Maryland, March 27 - 30, March 2000.

[77] Byoung-Dai Lee, Jon Weissman. An Adaptive Service Grid Architecture Using Dynamic Replica Management, *International Grid Computing Workshop (Grid2001)*, Denver, Colorado, November 12, 2001.

[78] Hyeran Lim, Vikram Kapoor, Chirag Wighe, David Du. Active Disk File System: A Distributed, Scalable File System, *18th IEEE Symposium on Mass Storage Systems and 9th NASA Goddard Conference on Mass Storage Systems and Technologies*, San Diego, April 17-20, 2001.

[79] Joshua P. MacDonald, XDelta, http://www.XCF.Berkeley.EDU/*sim*jmacd/xdelta.html

[80] Monarc Project: Models of Networked Analysis at Regional Centres for LHC Experiments, http://monarc.web.cern.ch/MONARC, January 2000.

[81] Priya Narasimhan, Luise Moser, P.M. Melliar-Smith. Using Interceptors to Enhance CORBA, *IEEE Computer*, July, 1999.

[82] Anh Nguyen-Tuong, Andrew Grimshaw, M. Hyett. Exploiting Data-Flow for Fault-Tolerance in a Wide-Area Parallel System. *15th International Symposium on Reliable and Distributed Systems*, IEEE Computer Society, Niagara-on-the-Lake, Ontario, Canada, October 23-25, 1996.

[83] Newsletter at National University of Singapore, Novel Framework Enhances Web Performance and Scalability, Volume 5 Number 2, Apr./Jun. 2000.

[84] Matthias Nicola, Matthias Jarke. Increasing the Expressiveness of Analytical Performance Models for Replicated Databases, *International Conference on Database Theory (ICDT'99)*, Jerusalem, January 1999.

[85] D. Menasce, V. Almeida, L. Dowdy. Capacity Planning and Performance Modelling. Prentice Hall, New Jersey 1994.

[86] Brian Noble, Ben Fleis, and Minkyong Kim. A Case for Fluid Replication. *Network Storage Symposium (Netstore)*, October 1999.

[87] Object Management Group Inc., CORBA: http://www.corba.org

[88] ObjectStore http://www.exceloncorp.com/products/objectstore.html, June 2000.

[89] Objectivity Inc., http://www.objectivity.com

[90] Objectivity/FTO and Objectivity/DRO Part, Number: 60-FTODRO-0 Release 6.0, August 29, 2000.

[91] Ron Oldfield. Summary of existing and developing data grids. White paper for the Remote Data Access group of the Global Grid Forum, March 2001.

[92] Orcale Cooperation, http://www.oracle.com

[93] Esther Pacitti, Pascale Minet, Eric Simon. Replica Consistency in Lazy Master Replicated Databases. *Distributed and Parallel Databases*, Volume 9, Number 3, May 2001.

[94] Thomas Page, Richard Guy, John Heidemann, David Rathner, Peter Reiher, Ashvin Goel, Geoffrey Kuenning, Gerald Popek. Perspectives on Optimistically Replicated, Peer-to-Peer Filing. *Software-Practice and Experience'*, Volume 28, Number 2, February 1998.

[95] Particle Physics Data Grid project (PPDG): http://www.ppdg.net

[96] Karin Petersen, Mike Spreitzer, Douglas Terry, Marvin Theimer, Alan Demers. Flexible Update Propagation for Weakly Consistent Replication, *Sixteenth ACM Symposium on Operating System Principles*, St. Malo, France, October 5-8, 1997.

[97] Jon Postel, Joyce Reynolds, File Transfer Protocol (FTP), Request For Comments (RFC) 959, October 1985.

[98] Michael Rabinovich, Narain Gehani, Alex Kononov. Scalable Update Propagation in Epidemic Replicated Databases. *5th International Conference on Extending Database Technology*, Lecture Notes in Computer Science 1057 Springer 1996, Avignon, France, March 25-29, 1996.

[99] Michael Rabinovich, Irina Rabinovich, Rajmohan Rajaraman, Amit Aggarwal. A Dynamic Object Replication and Migration Protocol for an Internet Hosting Service. *19th International Conference on Distributed Computing Systems*, IEEE Computer Society, Austin, Texas, May 31 - June 4, 1999.

[100] Kavitha Ranganathan, Ian Foster. Identifying Dynamic Replication Strategies for a High Performance Data Grid, *International Grid Computing Workshop (Grid2001)*, Denver, Colorado, November 12, 2001.

[101] RD45: A Persistent Object Manager for HEP, http://wwwinfo.cern.ch/asd/rd45/index.html

[102] RD45 Project. Wide-Area Objectivity/DB Federations, White paper, CERN/RD45/98/05, July 1998.

[103] RD45 Project. ODBMS Experience - Object Databases and HEP: What Have We Learnt So Far?, White paper, CERN/RD45/99/07, March 1999.

[104] Erik Riedel, Susan Spence, Alistair Veitch. When Local Becomes Global: An Application Study of Data Consistency in a Networked World, *20th IEEE International Performance, Computing and Communications Conference (IPCCC2001)*, Phoenix, Arizona, April 4-6, 2001.

[105] L. Roberts, Beyond Moore's Law: Internet Growth Trends, IEEE Computer, January 2000.

[106] Root object-oriented Analysis framework, http://root.cern.ch

[107] L. Rurua. Possibilities for h0 → b-bbar with CMS at LHC, 5th International Conference on B-Physics "Beauty 97" Hadron Machines, Los Angeles, California, October 1997.

[108] 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.

[109] R. Sandberg. The Sun Network File System: Design, Implementation and Experience, Tech. Report, Mountain View CA: Sun Microsystems, 1987.

[110] M. Satyanarayanan, James Kistler, Ellen Siegel. Coda: A Resilient Distributed File System, *IEEE Workshop on Workstation Operating Systems*, Cambridge, MA, November 1987.

[111] Luciano Serafini, Heinz Stockinger, Kurt Stockinger, Floriano Zini. Agent-Based Query Optimisation in a Grid Environment, *IASTED International Conference on Applied Informatics (AI2001)*, Innsbruck, Austria, February 19 -22, 2001.

[112] SETI@home: Search for Extraterrestrial Intelligence, http://setiathome.ssl.berkeley.edu, 2001.

[113] Jeff Sidell, Paul M. Aoki, Adam Sah, Carl Staelin, Michael Stonebraker, Andrew Yu. Data Replication in Mariposa, *Twelfth International Conference on Data Engineering*, IEEE Computer Society , New Orleans, Louisiana, February 26 - March 1, 1996.

[114] Stanford Linear Accelerator Center (SLAC), http://www.slac.stanford.edu

[115] Ioana Stanoi, Divyakant Agrawal, Amr El Abbadi. Using Broadcast Primitives in Replicated Databases, *International Conference on Distributed Computing Systems (ICDCS '98)*, IEEE Computer Society, Amsterdam, Netherlands, May 26 - 29, 1998.

[116] Heinz Stockinger, Data Replication in Distributed Database Systems, CMS Note 1999/046, July 1999.

[117] Heinz Stockinger. Distributed Database Management Systems and the Data Grid. *18th IEEE Symposium on Mass Storage Systems and 9th NASA Goddard Conference on Mass Storage Systems and Technologies*, San Diego, April 17-20, 2001.

[118] Heinz Stockinger, Omer F. Rana, Reagan Moore, Andre Merzky. Data Management in a Grid Environment, *European High Performance Computing & Networking Conference (HPCN2001)*, Amsterdam, The Nederlands, June 25 - 27, 2001.

[119] Heinz Stockinger, Kurt Stockinger, Erich Schikuta, Ian Willers. Towards a Cost Model for Distributed and Replicated Data Stores, *9th Euromicro Workshop on Parallel and Distributed Processing (PDP 2001)*, IEEE Computer Society Press, Mantova, Italy, February 7-9, 2001.

[120] 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 (HPDC-10)*, San Francisco, California, August 7-9, 2001.

[121] Heinz Stockinger, Andrew Hanushevsky. HTTP Redirection for Replica Catalogue Lookups in Data Grids, to appear in *ACM Symposium on Applied Computing (SAC2002)*, Madrid, Spain, March 10-14, 2002.

[122] Heinz Stockinger, Asad Samar, Shahzad Muzaffar, Flavia Donno, Andrea Domenici. Grid Data Mirroring Package (GDMP): User Guide for GDMP 2.0, http://cmsdoc.cern.ch/cms/grid, October 2001.

[123] Kurt Stockinger, Dirk Düllmann, 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, Springer-Verlag, Sept. 2000.

[124] Supercomputing Conferences: http://www.sc2000.org and http://www.sc2001.org

[125] Andrew Tanenbaum, Albert Woodhull. Operating Systems: Design and Implementation, 2nd Edition, Prentice Hall, 1997.

[126] Sudharshan Vazhkudai, Steve Tuecke, Ian Foster. Replica Selection in the Globus Data Grid, *IEEE International Conference on Cluster Computing and the Grid (CCGRID 2001)*, Brisbane, Australia, May 16 - 18, 2001

[127] Versant Inc., http://www.versant.com/, December 2001.

[128] Edgar Weippl, Wolfgang Essmayr. Fine Grained Replication in Distributed Databases: A Taxonomy and Practical Considerations. *Database and Expert Systems Applications (DEXA 2000)*, London, UK, September 4-8, 2000.

[129] Gio Wiederhold. Mediators in the Architecture of Future Information System. *IEEE Computer*, 25(3):38-49, March 1992.

[130] Matthias Wiesmann, Andr Schiper, Fernando Pedone, Bettina Kemme, Gustavo Alonso. Database Replication Techniques: A Three Parameter Classification. *19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, Nürnberg, Germany, October 16-18, 2000.

[131] Tony Wildish. Accessing Objectivity catalogues via the web. http://wildish.home.cern.ch/wildish/Objectivity/scripts.html

[132] Ian Willers. CMS Interim Memorandum of Understanding The Costs and How They are Calculated, *CMS Note 2001/035*, July 2001.

[133] Ouri Wolfson, Sushil Jajodia, Yixiu Huang. An Adaptive Data Replication Algorithm, *ACM Transactions on Database Systems (TODS)*, Volume 22, Number 2, June 1997.

[134] Rich Wolski, Neil Spring, Jim Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing, *Journal of Future Generation Computing Systems*, 1998.

[135] Avishai Wool. Quorum Systems in Replicated Databases: Science or Fiction? Bulletin of the IEEE Computer Society Technical Committee on Data Engineering. 1998.

[136] Wengyik Yeong, Tim Howes, Steve Kille. Lightweight Directory Access Protocol, Request For Comments (RFC) 1777, March 1995.