On Data Placement Issues in Grid Computing Environments

Michael Höfling

November 2008
Master’s Thesis in Computing Science, 30 ECTS credits
Supervisor at CS-UmU: Johan Tordsson
Examiner: Per Lindström

UMEÅ UNIVERSITY
DEPARTMENT OF COMPUTING SCIENCE
SE-901 87 UMEÅ
SWEDEN
Abstract

While Grid job scheduling has received much attention in the recent years, relatively few researchers have studied data placement issues. Although job management is important in Grid computing, data management and placement is likely to be among the most challenging issues for future Grid applications. The current job management frameworks by the Umeå Grid Infrastructure Research and Development group lack data management capabilities. The Data Placement Service, an easy to use and integrate service for data placement for Grid environments, was developed in an earlier project to investigate the feasibility of an implementation of the missing capabilities.

This thesis presents and discusses data placement issues that are common in the context of Grid computing and distributed data storage, e.g., storage discovery, storage allocation, data replication, data consistency control, reliable file transfers, job-aware data placement optimization, and transactions. Solutions to selected issues are implemented as a proof-of-concept in the Data Placement Service as a practical part of the thesis. Existing data management systems are surveyed and their capabilities are compared with that of the DPS. The thesis is concluded with a discussion of future directions of this work.
# Contents

List of Figures vii  
List of Abbreviations ix  

1 Introduction 1  

2 Problem Description 3  

2.1 Environment and Preconditions ........................................ 3  
2.2 Problem Statement .......................................................... 3  
2.3 Goals ........................................................................... 3  

3 Technological and Architectural Background 5  

3.1 Grid Computing ................................................................. 5  
3.2 Service-Oriented Architecture ........................................... 6  
3.2.1 Services ..................................................................... 6  
3.2.2 Loose Coupling ............................................................ 6  
3.3 Web Services .................................................................. 7  
3.3.1 Web Service Description Language .................................. 8  
3.3.2 Simple Object Access Protocol ..................................... 8  
3.3.3 Web Service Resource Framework ................................. 8  
3.3.4 Implementation of Web Services .................................. 10  
3.3.5 Web Service Usage ....................................................... 10  
3.4 Globus Toolkit 4 ................................................................. 11  
3.4.1 Architecture ................................................................. 11  
3.4.2 Components ................................................................. 12  
3.5 Data Placement Service ..................................................... 16  
3.5.1 Architecture ................................................................. 16  
3.5.2 Components ................................................................. 16  
3.5.3 DPS Component Interaction ......................................... 18  
3.5.4 Limitations ................................................................. 19
4 Data Placement Issues
4.1 Storage Discovery .......................... 21
4.2 Storage Allocation ......................... 23
4.3 Data Replication ......................... 25
4.4 Data Consistency Control ............... 25
4.5 Multi-Protocol Reliable File Transfer .... 26
4.6 Job-Aware Data Placement Optimization ... 28
4.7 Transactions ............................. 30

5 Implementation ......................... 31
5.1 Data Replication .......................... 31
5.2 Data Consistency Control ............... 32
5.3 Dynamic Storage Discovery ............ 33
5.4 Reliable File Transfer ................... 34

6 Discussion ............................. 37
6.1 Related Work ............................ 37
6.2 Future Work ............................. 39

7 Acknowledgements ..................... 41

References .............................. 43

A DPS Interfaces ......................... 47
A.1 StorageBroker Interface ................. 47
A.2 Replicator Interface .................... 48
A.3 StorageElementHandler Interface ........ 48

B Installing the Data Placement Service .... 49
B.1 Requirements ............................ 49
B.2 Download Software ..................... 49
B.3 Configure the DPS .................... 49
B.4 Install the DPS ......................... 54
B.5 Use the DPS ............................ 54

C Public Interface of the Data Placement Service .... 55
C.1 Client Library ............................ 55
C.2 Command Line Interface ................ 55

D Sample Command Line Tools Output .... 57
D.1 Upload a File ........................... 57
D.2 Verify a File ............................ 58
D.3 Replicate a File ......................... 59
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.4</td>
<td>Download a File</td>
<td>61</td>
</tr>
<tr>
<td>D.5</td>
<td>Remove a File</td>
<td>62</td>
</tr>
</tbody>
</table>
List of Figures

3.1 The publish, find, bind pattern enables loose coupling of service consumers and service providers through service brokers in an SOA. .......................... 7
3.2 A network stack model of a WS. ................................................. 8
3.3 The resource approach to stateful WS. Taken from [30]. ......................... 9
3.4 A WS-Resource. Taken from [30]. .............................................. 9
3.5 GT4 service overview. Taken from [18]. ......................................... 11
3.6 Example of a delegation of user credentials. Taken from [30]. .................. 12
3.7 A proxy certificate. Taken from [30]. ............................................ 13
3.8 Example Replica Location Service configuration. Taken from [14]. .......... 14
3.9 Information flow in the MDS. Taken from [1]. .................................. 15
3.10 Overview of the DPS architecture. ................................................. 17
3.11 Sequence diagram of the storage discovery and selection process. .......... 17
3.12 Sequence diagram for the file upload and catalog registration process. ... 18

4.1 Storage discovery with broadcast messages. ...................................... 22
4.2 Storage discovery with central SE index. ......................................... 23
4.3 Soft storage allocation. ................................................................. 24
4.4 Basic reliable file transfer. ............................................................ 26
4.5 Multi-protocol reliable file transfer using a transfer cache. ................. 27
4.6 Job-aware data placement optimization as proposed in [29]. ............... 29

5.1 Sequence diagram for data replication. .......................................... 32
5.2 Sequence diagram for client-side data consistency check. ................... 33
5.3 Publication of SE availability in a hierarchical index. ........................ 34
5.4 Finite-state machine for reliable file transfer. .................................. 35
5.5 Sequence diagram of the SE handler dispatcher. ................................ 35
List of Abbreviations

ACID ............ Atomicity, Consistency, Isolation, Durability, page 30
API ............. Application Programming Interface, page 12
DPS ............. Data Placement Service, page 16
FSM ............. Finite-State Machine, page 34
GridFTP .......... Grid File Transfer Protocol, page 13
GSI ............. Grid Security Infrastructure, page 12
GT4 ............. Globus Toolkit 4, page 11
HTML ............ HyperText Markup Language, page 7
HTTP ............ HyperText Transfer Protocol, page 8
LFN ............. Logical File Name, page 14
LRC ............. Local Replica Catalog, page 14
MDS ............. Monitor and Discovery Service, page 14
OASIS ............ Organization for the Advancement of Structured Information Stan-
dards, page 6
OGF ............. The Open Grid Forum, formerly known as Global Grid Forum, page 11
OGSA ............ Open Grid Services Architecture, page 11
PFN ............. Physical File Name, page 14
REST ............ Representational State Transfer, page 10
RFT ............. Reliable File Transfer, page 13
RLI ............. Replica Location Index, page 14
List of Abbreviations

RLS ............... Replica Location Service, page 14
RPC ............... Remote Procedure Call, page 10
SE ............... Storage Element, page 14
SOA ............... Service-Oriented Architecture, page 6
SOAP ............... Simple Object Access Protocol, page 8
VO ............... Virtual Organization, page 6
WS-Notification . Web Service Notification, page 8
WS-Resource ..... Web Service Resource, page 8
WSDL .............. Web Service Description Language, page 8
WSRF .............. Web Service Resource Framework, page 8
WWW .............. World Wide Web, page 7
XML ............... eXtensible Markup Language, page 8
Chapter 1

Introduction

The Grid Infrastructure Research and Development (GIRD) [2] group’s current research focus is on Grid job management. A job in the scope of this thesis is a program that runs on a computer and uses data stored somewhere, e.g., on the Grid itself, as input. The GIRD group has experience in building services for resource brokering (selection of which resource should be used for a job) and in the construction of tools for automatic management of large numbers of jobs.

Data placement decisions are similar to job placement decisions. Both placement decisions may involve resource reservation under certain constraints and in a distributed manner. However, the types of resources, the used heuristics, and the reservation time for the respective placement decisions cannot be compared with each other. Data placement in Grid environments includes long-lasting storage of data, often permanent and read-only. Conversely, job placement includes ad-hoc reservations and usage of temporarily available computational resources.

There are systems that try to unify job and data management, but as already described, these are two different tasks to face in a Grid environment. Furthermore, these systems often result in huge centralized data centers or strongly coupled architectures. They violate the principle of a Grid and a Service-Oriented Architecture. The data placement issues discussed in this thesis and the presented solutions are applicable to a variety of distributed storage systems, and are especially well-suited for loosely-coupled system architectures.

This thesis is divided into two parts, a theoretical part discussing data placement issues that were identified in a previous thesis [23] and a practical part describing how these issues are solved in the current implementation, the Data Placement Service (DPS). The DPS is an easy to use and integrate service for data placement for Grid environments. Alongside the already existing functionality of the DPS, e.g., automatic selection of storage location with customizable selection heuristics, file name virtualization through a catalog service, and a client API for simplified interaction with the Grid, new functionalities have been added. These functionalities include data replication, dynamic storage discovery, reliable file transfer, and a facility to ensure data consistency within the Grid. Additionally, a command line interface has been added to enable easy user interaction with the system.

The remainder of the thesis is structured as follows. Chapter 2 describes the studied problem and defines the goals of this project. An introduction to Web Services, their basic technologies and Service-Oriented Architectures along with details on the compo-
ments of the Globus Toolkit 4 and a brief description of the DPS are given in Chapter 3. A discussion about data placement issues in Grid-Computing environments is found in Chapter 4. Details on the implementation of the then-found solutions in the Data Placement Service are given in Chapter 5. Chapter 6 concludes the thesis with a short evaluation of the work, a comparison to related research efforts and future directions of the work.
Chapter 2

Problem Description

This chapter gives a more detailed description of the purpose and goals of the present work. The first part introduces the environment and preconditions for the project whereas the second part introduces the problems that are discussed in the later chapters and solved by the implementation. Finally, the third part specifies the goals of the work.

2.1 Environment and Preconditions

Previous work [23] describes the development of the Data Placement Service (DPS), an easy to use and integrate service for data placement for Grid environments. Its features include automatic selection of storage location with customizable selection heuristics, file name virtualization through a catalog service, and a client API for simplified interaction with the Grid.

The Globus Toolkit 4 [17], a software toolkit for building Grid applications, is used as technological basis for the DPS and thus also serves as a basis for the implementation in this thesis.

2.2 Problem Statement

In a previous thesis [23] knowledge about data placement was gained and data placement issues were discussed. An in-depth study of these issues was, however, postponed for future work. Based on these findings the following data placement issues are identified: storage discovery, storage allocation, data replication, data consistency control, reliable file transfer, job-aware data placement optimization, and transactions.

2.3 Goals

Based on the preconditions and the problem statement, the goal of the thesis project is to perform an in-depth study of the identified problems. This includes the survey of different approaches to data consistency, an investigation of replication strategies, and an exploration of methods for dynamic storage discovery. Furthermore, job-aware data placement optimization through a combination of data placement and job scheduling strategies is discussed. Another emphasis of the thesis is an investigation of how to
achieve multi-protocol reliable file transfer. The knowledge gained in the in-depth study is used to enhance the DPS prototype.
Chapter 3

Technological and Architectural Background

This chapter introduces basic concepts and technologies that have been used for the development of the practical part of this thesis, the Data Placement Service reference implementation. The issues discussed in this thesis as well as the practical part of the thesis focus on Grid computing environments. Thus, the field of Grid computing is briefly introduced first. Web services and Service-Oriented Architecture, the underlying technologies and architectural paradigm for Grid computing are described thereafter. The Globus Toolkit, a middleware for Grid computing environments and basis for the practical part of the thesis, is also introduced. The chapter is concluded by a short presentation of the DPS.

3.1 Grid Computing

The name Grid computing originates from the comparison of a computational Grid with the power grid, which provides access to electricity on demand through wall sockets. Users do not have to concern themselves with how and where the electricity is coming from. In the overall Grid vision, a Grid is a system that provides access to computational resources on demand without requiring knowledge about how and where these resources are located. In 2002 Ian Foster, one of the pioneers in the field of Grid computing, created a three point checklist[19] that defines a Grid as a system that

1. coordinates resources that are not subject to centralized control . . .
2. . . . using standard, open, general-purpose protocols and interfaces . . .
3. . . . to deliver nontrivial qualities of service\textsuperscript{1}.

Grid users want to get direct access to computers, software, data or other resources, e.g., devices such as sensors [20]. To coordinate the usage of resources in a controlled way without a centralized point of control, individuals and institutes have to make up sharing rules, e.g., who is allowed to access resources and under what conditions sharing

\textsuperscript{1}Quality of service means that it is guaranteed that a service is available under certain agreements, e.g., the guaranteed 99.999% availability of the publicly-switched telephone network in Germany, also known as five-nines.
is acceptable. A set of individuals and/or institutes defined by such sharing rules forms a Virtual Organization (VO) [20]. The term virtual implies that the organizations can internally consist of different physical participating and contributing organizations and resources. VOs can also be viewed as allocations of computing and storage resources. The resources to be coordinated are typically computing resources, e.g., PCs, workstations, servers, storage elements, etc. For communication with the Grid, as well as within in the Grid, standard, open, general-purpose protocols and interfaces, e.g., Web services, are used. These protocols increase the value of interconnecting resources. The nontrivial qualities of service that are provided by a Grid indicate that the Grid adds greater value than can be obtained from using the resources individually.

In practice, a Grid is built on top of a heterogeneous computing infrastructure, e.g., specialized hardware and commodity computers are combined to form a collaborative infrastructure. To utilize these heterogeneous resources and enable interoperability between them, the principles of a Service-Oriented Architecture in combination with Web services are often used. Both these paradigms are described in the following sections.

3.2 Service-Oriented Architecture

The concept of a Service-Oriented Architecture (SOA) defines a paradigm for building distributed systems on the basis of loosely coupled, distributed services. The Organization for the Advancement of Structured Information Standards (OASIS) defines a SOA as follows [6]:

Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains.

3.2.1 Services

The central part of a SOA is, as implied by the name, the service. In an SOA, the functionalities of a system or the components of a system are grouped and packaged as services. A service exposes its functionality over the network to enable it to be used in distributed environments. Services offer their functionality to service consumers through predefined interfaces. Services can communicate with each other and services can coordinate their activities. A service can provide simple, atomic functionality or it can be re-used and combined with other services to form new, composite services. The process of combining several services to new services is called orchestration. Orchestration maps a process to a set of available services.

3.2.2 Loose Coupling

Every service provides a description of its abilities, which includes information on how to interact with the service, e.g., how to invoke the service and what the result of the invocation is. In doing this, services are loosely coupled, e.g., they need minimal knowledge about each other as every service is self-descriptive.

In legacy distributed systems the dependencies between the components are often hard-coded and version specific. This means, if a subcomponent is changed due to a software update, upper layer components might easily break. By the introduction of an SOA, the subcomponents are represented as services that provide a predefined interface. This interface should, under the assumption that the SOA is well designed, never undergo major changes. This means even if a subcomponent is updated or completely replaced
3.3 Web Services

Web services (WS) are a distributed computing technology that allow the creation of client/server applications. The World Wide Web Consortium (W3C) defines Web services as follows[31]:

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

This definition refers to technology that is commonly used to implement WS. However, XML and HTTP are not the only possibilities to implement WS. Figure 3.2 shows a stack model of a WS using the technologies indicated in the definition. Unlike applications and technologies designed for the interaction between humans and computers,
such as the World Wide Web (WWW) and the Hypertext Markup Language (HTML), WS are designed to be used for machine-to-machine interaction only.\(^2\)

### 3.3.1 Web Service Description Language

Web services expose only their interfaces to the public. The interfaces are described in a description language, the Web Service Description Language (WSDL). WSDL itself is a language based on the eXtensible Markup Language (XML) and is as such independent of platform and programming language. The service description defines how to invoke the WS, explains the operations offered by the WS as well as the data types that are used for operation requests and that are returned as operation responses. In WS terminology operations are called ports and the data that is exchanged are called messages.

### 3.3.2 Simple Object Access Protocol

While WSDL is the language describing the WS itself, the Simple Object Access Protocol (SOAP) is used for the actual interaction between the Web service and a client. SOAP is used for both operation invocation and data exchange. Like WSDL, SOAP is XML-based and is thus also platform and language-independent. Commonly, SOAP uses the HyperText Transfer Protocol (HTTP) as transport protocol but is independent of the underlying transport protocol, e.g., other protocols can be used as well.

### 3.3.3 Web Service Resource Framework

WSs are per se stateless, i.e., a WS cannot remember information, or keep state, from one invocation to another. While WSs do not need to be stateful generally, Grid applications do. In order to store and manipulate state in a standardized manner without breaking the stateful nature of WSs, the WS Resource Framework (WSRF) defines stateful resources as an extension to the WS standards. Figure 3.3 shows the resource approach to stateful WS. When a client invokes a stateful WS it has to specify both the WS and the resource it wants to use. The combination of WS and resource is called a WS-Resource. The idea behind this is shown in Figure 3.4. The information stored within a WS-Resource is called WS-ResourceProperties and is a set of typed values. On the client side, WSRF enables the client to read and manipulate the WS-ResourceProperties of the WS. The client can query a WS for WS-ResourceProperties or subscribe to the WS for notifications. For the latter, the WSRF standard includes a mechanism called WS-Notification which realizes the observer design pattern[22] for WS.

---

\(^2\)It is possible, if unlikely, for a human to use WS for human to computer interaction directly.
3.3. Web Services

Figure 3.3: The resource approach to stateful WS. Taken from [30].

Figure 3.4: A WS-Resource. Taken from [30].
3.3.4 Implementation of Web Services
As the definition of a WS interface is platform-independent and language-independent, the WS and the clients can be implemented in virtually any programming languages that supports WS, e.g., Java, C, C#, Ruby, or Python. Thus it is possible for the client and server side parts of a WS to be implemented in different languages as long as the common interface definition in WSDL is fulfilled. This enables, for example, rapid client development using Java while the server side is implemented in C for performance reasons or a server side implemented in Python for fast prototyping and then replaced by an improved and more robust implementation in Java or C.

3.3.5 Web Service Usage
There are, in general, three applications of WS, remote procedure calls (RPC), representational state transfers (REST) and SOA. RPC means that a client calls a procedure provided on a remote system. The client, the WS consumer, is communicating directly with the WS provider. Loose coupling is normally not necessary if WSs are used for RPC and as such not provided by the surrounding architecture. REST[16] is a style of software architecture for distributed systems, similar to a SOA, but not described in more detail here. WS can, however, be used to realize REST. The latter mentioned application of WS is SOA. WSs are a concrete rendering of a SOA, but by no means the only possible one.
3.4 Globus Toolkit 4

The last two sections introduced both WSs and SOAs as paradigms for building collaborative, distributed systems. Both paradigms offer technological bases for Grid computing infrastructure. However, the development of Grid applications and Grid infrastructure often includes a common set of problems to be solved.

Motivated by this, the Open Grid Forum (OGF)\cite{9} developed the SOA-based Open Grid Services Architecture (OGSA)\cite{8}. OGSA aims to define a common, standard and open architecture for Grid-based applications\cite{8, 30}. Functionalities described by OGSA include, e.g., security, resource management, job management, data services, VO management. The Globus Toolkit 4 (GT4) is a toolkit that provides tools and components for the development of Grid applications. It partially fulfills the vision of the OGSA. Some functionalities are missing, and others have different interfaces than those proposed by OGSA. The components are mostly structured as WS and form an SOA of loosely coupled, communicating services. GT4 is used as middleware in the software that has been developed as part of this thesis.

3.4.1 Architecture

Figure 3.5 gives an overview of the GT4 architecture. GT4 includes a set of services that form a common infrastructure for distributed computing applications. These services provide, e.g., security, resource management, resource discovery and data management. Most of these services are implemented in Java and expose their functionality as Web services to facilitate a SOA while other services are implemented in C and have custom protocols, e.g., the GridFTP server and the GridFTP protocol, further described in Section 3.4.2.2.
3.4.1.1 Web Service Runtime Environment

Web services that are meant to be used with GT4 run within Web service runtime environments known as *containers*. These containers implement SOAP over HTTP as message transport protocol and a number of other Web service specifications. GT4 offers Web service containers for Java, C and Python. Each container has a registry interface that allows clients to discover which services are running in the respective container. In GT4 terminology these containers are called *WS cores*.

3.4.1.2 Client APIs and Tools

The services in GT4 include client Application Programming Interfaces (APIs) for different programming languages, typically C and Java, that allow simplified interaction with the provided services. In addition, some services have command line tools for interactive or administrative purposes.

3.4.2 Components

The components and technologies of GT4 that are used in the thesis are briefly introduced in this section. Explaining GT4 in detail is, however, out of the scope of this thesis. There is available literature such as [7, 30, 24], which can be consulted by the interested reader for a more detailed explanation of GT4.

3.4.2.1 Security

Security is of particular importance in Grid computing environments as a Grid is not necessarily limited by organizational boundaries. To meet security needs, GT4 offers a flexible toolkit for authentication and authorization purposes that is supported in all parts of the framework, including both Web services and non-Web service components. This security system is called the Grid Security Infrastructure (GSI).

In GT4’s default configuration, each user and service is assumed to have a X.509 public key credential, e.g., a certificate. There are protocols that allow two entities to validate each other’s credentials in order to use these credentials to establish a secure channel for purposes of message protection and to create and transport delegated credentials, called proxy certificates, see Figure 3.7. Proxy certificates allow a remote component to act on a user’s behalf for a limited period of time, as illustrated in Figure 3.6. Authorization call outs associated with GT4 services can be used to determine whether specific requested operations should be allowed. Supporting tools, some in GT4 proper and some available from other sources, enable the generation, storage, and retrieval of the credentials that GT4 uses for authentication, and address issues concerning group membership etc.[18]

GT4 offers APIs to easily integrate GSI in new applications.
3.4.2.2 Data Management

GT4 provides a number of tools for data management including data transfer and data replication. Data transfer is supported by GridFTP and the Reliable File Transfer service while data replication is offered through the Replica Location Service and the Data Replication Service.

GridFTP. File transfers inside GT4-based Grids are normally performed using the Grid File Transfer Protocol (GridFTP). GridFTP is an extended version of the FTP protocol that is optimized for large, parallel data transfers. It supports authentication through X.509 certificates. Furthermore, it supports transfers of parts of files and third-party file transfers\(^3\). GridFTP is the de-facto standard file transfer protocol used in Grid computing environments. There are GridFTP command-line tools and APIs for several programming languages, e.g., for C and Java. An additional feature of the GridFTP protocol is markers that are sent from the receiving server to the transfer controlling client, e.g., the sending server or the controlling client. Currently, there are two different marker classes, the performance markers and the restart markers. Performance markers can be used to measure the transfer’s performance. Restart markers tell the client how much of the file has been received by the server. If the transfer fails, the client may utilize the partial transfer mechanism to restart the transfer by providing the markers, and the transfer resumes from where it left off. This mechanism enables reliable file transfer.

Reliable File Transfer. The GridFTP protocol itself does not offer reliable file transfer, but, as described earlier, offers features to implement such a functionality as a high-level service. The Reliable File Transfer (RFT) service uses features of the GridFTP protocol to expose reliable file transfer as a WS in GT4. A file transfer job consists of a list of URLs, e.g., transfer sources and transfer sinks, a job state and additional configuration settings for the GridFTP transfer. RFT stores information about active transfers in a database and is hence able to recover failed or interrupted transfers. Notably, RFT only supports third-party transfers between servers and not reliable file transfer from a client to a server.

\(^3\)Third-party file transfer means server-to-server file transfer without relaying the data through a client.
Replica Location Service. File replication is the process of storing multiple copies of the same file at different physical locations. The gained redundancy improves reliability, fault-tolerance and accessibility. The copies are called replicas. A URL pointing to a physical copy of the file is called a Physical File Name (PFN) of the file. The set of PFNs is mapped to a system-wide unique identifier, called the Logical File Name (LFN). The purpose of this mapping is to simplify the identification of file in a data Grid. Furthermore, this mapping allows the creation of a logical file namespace, e.g., file name virtualization. GT4 offers a service to manage these mappings in a registry called the Replica Location Service (RLS). The RLS system consists of three components, the Local Replica Catalog (LRC), the Replica Location Index (RLI), and the RLS itself. The LRC stores the mapping between LFNs and PFNs for an area, e.g., one Storage Element or a group of Storage Elements. A Storage Element (SE) provides uniform access to storage resources, e.g., a disk server, an array of disk servers, or a mass storage system. The RLI stores the mapping between the LFNs and the LRCs that store information about the LFNs, e.g., file attributes and PFNs. LRCs are registered to at least one RLI and send updates to these if mappings change. RLIs can also communicate with other RLIs. The RLS interfaces both LRC and RLI to query them for information. By default, RLS offers simple mapping between logical file names (LFNs) and physical file names (PFNs) of files. In addition to the LFN-PFN mapping, RLS also supports custom attributes in both LFN and PFN, e.g., file size and checksum. Figure 3.8 shows a sample configuration of the RLS. In the 4.0.x releases of GT4, RLS has a proprietary communication protocol that can be used through APIs in Java and C. This changed with the release of the 4.2.x series of GT4, where RLS exposes its functionality through a WS interface.

3.4.2.3 Information Services

GT4 provides services to monitor and discover arbitrary types of Grid resources. These services are called the Monitor and Discovery Service (MDS). Even though the name implies a single service, the MDS framework exposes its functionalities through three services: the Index Service, the Trigger Service and the Archive Service. Figure 3.9 gives an overview of the information flow in the MDS. MDS allows users to discover what resources are part of a VO and monitor those resources. Additionally, the MDS provides query and subscription interfaces to retrieve arbitrarily detailed resource data through the Index Service, and the Trigger Service that can be configured to take action when pre-configured conditions are met. The Archive Service, shown in Figure 3.9, is still in development and is not described further as it is of no interest to the thesis.
3.4. Globus Toolkit 4

The MDS services are based on the Aggregator Framework that is used to build services that collect and aggregate data. Information is collected by Aggregator Sources, aggregated, and then published to Aggregator Services. MDS includes three Aggregator Sources: the Query Aggregator Source, the Subscription Aggregator Source and the Execution Source, as shown in Figure 3.9. The Query Aggregator Source retrieves its information by polling a WSRF service for its WS-ResourceProperty information. The Subscription Aggregator Source retrieves its information by subscribing to a WSRF-based service and then receiving updates through WS-Notification. The third type of Aggregator source is the Execution Source that executes a program on the computer running MDS to collect information and publish it in the MDS.

Index Service. The Index Service is a registry that a client can use to discover information that has been aggregated using the previously mentioned sources. Clients can use WSRF resource property query or the subscription/notification interface to retrieve information from an Index. Indices can register to each other in a hierarchical fashion in order to aggregate information from multiple Indices at several levels. The information stored in an Index has a lifetime and is removed from the Index if it is not refreshed, making the Index self-cleaning.

Trigger Service. The Trigger Service collects information from the Aggregator Sources and compares this data against a set of pre-defined conditions. When a condition is met, an action is triggered. Potential use cases for this functionality include the replication of a file if a storage element becomes unavailable and the accessibility of the file is thus endangered.
3.5 Data Placement Service

The Data Placement Service (DPS), earlier developed as part of [23], is an easy to use and integrate service for data placement for Grid environments. It supports automatic selection of storage location with customizable selection heuristics, file name virtualization through a catalog service, and a client API for simplified interaction with the Grid. The DPS includes two services: the CatalogService and the StorageBrokerService. The primary interface between the users and these services is the StorageBroker client. In this section the architecture and functionality of these components are described, followed by an overview of the logical flow and a discussion of the limitations of the first DPS version.

3.5.1 Architecture

The DPS is based on WS and implemented entirely in Java. All services have been developed and deployed in the GT4 Java WS core. The communication between the client and the services takes place via the HTTP(S) protocol and the data format of the messages is XML, with the request being wrapped using standard SOAP RPC. The communication between the client and the storage elements takes place through the FTP and GridFTP protocols. The CatalogService data is stored in a relational database management system. The service has been tested and deployed with PostgreSQL as database back-end, using an abstract interface to enable the easy use of other relational databases, e.g., Oracle, MySQL.

3.5.2 Components

The DPS system components can be divided into two different groups: components on the Grid side and components on the client side. For the purpose of clarity, not all parts of the system are shown in Figure 3.10. The important components on the Grid side are the CatalogService, the StorageBrokerService and the Storages. Storages are file servers that are located in the Grid, e.g., FTP or GridFTP servers. The central component on the client side is the StorageBroker. In the following descriptions, including the figures, the term client is used for either a user or a program using the DPS.

3.5.2.1 CatalogService

The CatalogService is a registry that keeps track of where the files are stored in the Grid. It stores mappings between the logical file names (LFNs) and the physical file names (PFNs) of a file. Additionally, the CatalogService stores meta information about the file, e.g., a consistency checksum. All files that have been placed in the Grid using DPS are registered in the CatalogService. A client can query the CatalogService to get the physical locations of a file, or any associated metadata.

3.5.2.2 StorageBrokerService

The StorageBrokerService is a broker that enables the discovery of storages that are located in the Grid. Furthermore, it allows the filtering of discovered storages according to pre-defined heuristics. The discovery process itself is handled by the StorageDiscovery component whereas the storage selection is handled by the StorageSelection component. Figure 3.11 shows a sequence diagram of the storage discovery and selection...
Figure 3.10: Overview of the DPS architecture.

Figure 3.11: Sequence diagram of the storage discovery and selection process.
process. First, the client invokes the \texttt{getStorage()} method of the \texttt{StorageBrokerService} specifying the required space constraints. The \texttt{StorageBrokerService} next invokes the \texttt{StorageDiscovery} to get the storages and filters them using \texttt{StorageSelection} component. In the end, the filtered list of storages is returned to the client. Note that the client shown in Figure 3.11 need not correspond to the Client in Figure 3.10, e.g., the \texttt{StorageBrokerService} can also be invoked directly by a user or by other programs.

3.5.2.3 StorageBroker

The \texttt{StorageBroker} resides on the client side and acts as a mediator between the DPS components on the Grid side and the Client component on the client side. The \texttt{StorageBroker} coordinates all necessary steps for the desired operation, e.g., upload of a file to the Grid.

3.5.3 DPS Component Interaction

As an example of the logical flow of the DPS, the sequence diagram in Figure 3.12 illustrates the steps the \texttt{StorageBroker} component coordinates to upload a file to the Grid. The \texttt{UUIDGen} is a universal unique identifier generator from the Apache Axis project. It is used to generate the logical file names that are used inside the DPS. Hash is a file checksum generator that is used to calculate the file checksum for the file that should be uploaded. The Uploader component is a simplified storage access interface. It uploads a file to an FTP or GridFTP server using the Java GridFTPClient class that is provided by the GT4.

If a Client wants to upload a file to the Grid using the DPS it calls the \texttt{uploadToGrid()} method of the \texttt{StorageBroker}. The \texttt{StorageBroker} then generates a logical file name for the file using the \texttt{UUIDGen} and calculates the checksum using the Hash component. Next, the \texttt{StorageBroker} invokes the \texttt{StorageBrokerService} to discover storages in the
Grid. The returned storage list is used by the Uploader to upload the file to the Grid. In the end, before returning the logical file name to the Client, the LFN, the PFNs and its meta information, e.g., the checksum of the file, are registered in the CatalogService. The Client can now use the returned LFN to query the CatalogService for PFNs of the uploaded file. These PFNs can be used in, e.g., a job description submitted to the Grid.

3.5.4 Limitations

The first version of the DPS has been implemented as a proof-of-concept with limited functionalities, e.g., only file upload is supported by the StorageBroker. Other operations, e.g., file download, file replication, or file consistency checks, are not implemented but are possible to add to the architecture.

The Uploader only supports FTP and GridFTP as transfer protocols. Even though both protocols can be used to transfer files to the DPS, neither of them can guarantee reliable file transfer. Moreover, as there is no standardized way to determine free space on FTP and GridFTP servers, the DPS is unable to determine the free space on the storages. Instead, the StorageDiscovery uses the FTP \texttt{ALLO}\textsuperscript{[28]} command for space allocation on the respective servers on storage discovery. Support for this command in the FTP server implementation seems, however, to be optional. Another shortcoming is that the user has to know the LFN to find the PFNs of the file as the CatalogService does not support directory listing. Furthermore, when a file is no longer needed, it has to be deleted manually. The CatalogService entries can be removed by users through a special method. However, removing files from Storages by hand can lead to CatalogService inconsistencies.
Chapter 4

Data Placement Issues

Data placement issues are mostly a result of the distributed nature of the Grid and arise from the commonly used functionalities offered by data placement services. The herein discussed issues are storage discovery, storage allocation, file name virtualization, data replication, file consistency control, reliable file transfer, job-aware data placement optimization and ensuring system state consistency. The issues and their possible solutions are formulated without loss of generality to make it possible to adapt the ideas to other systems than the DPS. In the following sections the term user refers to a human user of the system whereas client refers to either a human user or a client application.

4.1 Storage Discovery

In a typical Grid environment all resources, e.g., computing nodes, storage nodes and broker nodes, have a dynamic life-cycle. They can become available at virtually any time and disappear without notice. With special focus on data placement, the resources of interest are the SEs. The problem is known as storage discovery.

There are at least two possible methods that are usable on both Grid and user side to discover storage resources. One option is to implement dynamic storage discovery using broadcast messages. A broadcast message is a message that, once sent to the network, is received by every device on the network. Figure 4.1 illustrates how the discovery process works. The StorageBrokerService (SBS) in the figure offers clients the functionality to discover storages that are available in the Grid for data placement. Notably, a data placement service need not necessarily have a service called SBS, but this role simplifies the illustration of the discovery process. In Step 1, the client sends a storage request to the SBS. The SBS sends a broadcast message to the network in Step 2. As a result of the properties of broadcast, all connected computers receive the broadcast message, in Step 3, and can react to it accordingly. Computational Nodes (CN) can ignore the broadcast message and not respond. In case the receiving computer is a SE, it returns a message in Step 4 including storage information. This response message might include information about the SE, e.g., available storage space, the supported transfer protocols, and the SE’s address. The aggregated information about the available SEs is pre-processed by the StorageBrokerService and then returned to the client in Step 5. There is no need for a central SE index as each client discovers the available SEs. Each client has the possibility to cache the discovery responses for a certain amount of time to avoid network congestion. Even though this solution has no single point of failure
it is unfeasible as not all computer networks support broadcasting. A solution for this may be the usage of multicast instead of broadcast, e.g., all resources of a Grid join a multicast group. Next, a client needs to send a multicast message to the multicast group and all Grid resources receive the message. However, this solution is also unfeasible as this would require multicast-aware routers in the network, e.g., multicast messages have to be exchanged between all participating subnetworks and thus adds more complexity to the network’s routing and security configuration. This excludes the typical virtual organization, e.g. Grid environment. The solution offers poor scalability as the SBS has to wait longer for discovery responses with increasing number of SEs. A broadcast-based or multicast-based solution might only work for smaller workgroups located on the same local network.

An alternative solution to dynamic storage discovery is to use a central SE index. Such a discovery scenario is shown in Figure 4.2. Here, all SEs publish their system information to the central index, as illustrated by the asterisk symbol in the figure. Much like the previous approach, the client sends a request to the SBS in Step 1. The SBS then polls the index and can pre-select SEs given constraints such as free space in Step 2. SE administrators have the possibility to define constraining policies and publish them in the central index alongside the SE information. Possible policies include, e.g., the maximum size of a single file, the maximum amount of data a user is allowed to upload, restrictions on allowed users, restrictions on allowed file types etc. After the SBS has gathered all necessary information and filtered it, the information is returned to the client in Step 4. Central index services are quite common in Grid environments even though they are considered to be a potential single point of failure. The primary advantage of storage discovery with a central index service is that it is easy to deploy in existing Grid infrastructures. The scalability of this solution is better than the scalability of the broadcast-based solution as the indices can be arranged in a hierarchical structure, but limited through the central index at the same time.
4.2 Storage Allocation

The motivating scenario for storage allocation is similar to the one for storage discovery. A user wants to upload one or more files to a set of SEs. In order to do this there has to be enough space on the SEs. As multiple users can upload files to an SE concurrently, the storage may become full, even though all users assert that free space is available before starting their uploads. The problem is that the space requirements are not met during the whole file transfer and that the aggregated space requirements for multiple users could not be fulfilled by the SE. Therefore, it should be possible to allocate space for a file transfer. This should be possible both on a per file basis and a bulk basis. The latter is to allocate storage space for a set of files instead of allocating once per file.

Soft space reservations are one approach to solve the storage allocation problem. A soft space reservation is represented by a soft-state, e.g., no space is physically reserved on the SE. If a user reserves space on a specific SE, a soft-state with the amount of space to reserve and period of validity is created. After this period of time, the space reservation is removed unless the reservation is renewed. These soft-state reservations have to be handled by a Soft Space Reservation Manager (SSRM) that can be deployed at the SE. The allocation and reservation process is illustrated in Figure 4.3. SEs have to register themselves at the SSRM with their storage space information and keep this information up-to-date through periodic updates, as illustrated by the asterisk symbol. In Step 1, the client reserves space by sending a request message to the SSRM identifying themselves and specifying the desired amount of space. The SSRM holds an internal list of active space reservations. Based on this information, the SSRM calculates the available storage space and decides whether the client’s request can be granted. If the client’s request cannot be fulfilled, the client receives an error message indicating this. Conversely, if the client’s request can be fulfilled, the SSRM creates a soft space reservation for the user. As shown in Figure 4.3, this reservation includes user name, amount of space and an expiration date. The client receives the reservation information in Step 2 and can (in Step 3) start to transfer the file(s) to the SE. As the client is aware of the expiration date of the space reservation, it has to update the reservation periodically in Step 4.
An advantage of soft space reservations is, that the reserved space on the SE is freed automatically in case of a client failure because of the period of validity. If a client crashes, or simply forgets to release its reservation, it is released as well. In this way, lazy reservations can be realized. The term lazy reservation means that the user reserves space on the SE, uploads the data to the SE and forgets to release the reservation. Two problems that might occur when using soft-states to represent space reservations and allocations are virtually full storages and malign or crashing clients. The problem of virtually full storages is easy to comprehend. If users repeatedly reserve storage space on a certain SE without actually using it, e.g., without uploading files, the SSRM states that the SE is full after some time. In the most extreme form the SE is not used at all in the physical sense, but the SSRM states that all storage space is reserved. This situation should, however, be handled by the expiration date that is defined for every storage space allocation. The second problem with malign or crashed clients is harder to prevent. In order for the soft space reservations to work, all space allocation requests, both from Grid clients and local users, have to pass through the SSRM. The consequence of this constraint can be reduced by integrating the SSRM in a server daemon software for example. The server daemon has to query the SSRM before it grants upload requests.

Donno et al. [15] describes how storage allocation in the Storage Resource Manager (SRM) is modeled. The SRM is an open standard that allows Grid middleware tools to communicate with site specific storage fabrics[11]. It supports storage allocation in a similar way to the one described here. However, storage allocation with SRM is possible in a more fine-grained way, e.g., a user requests storage space and the system returns the total reserved space and the guaranteed space. In addition to this two-level classification, the user has the possibility to include quality-of-service parameters in its request, e.g., retention policy and access latency requirements.
4.3 Data Replication

An SE can become unavailable due to, e.g., planned maintenance or crashes. As a result, all files provided by the SE become unavailable to Grid users and jobs. In order to prevent loss of data and reduce the impact of SE failure in the Grid, copies of files can be stored on multiple SEs. The storage of physical copies of a file on several SEs is known as data replication. In addition to the additional fault tolerance, data replication enables load balancing between the SEs when the same set of files is requested by users and jobs. It should be noted that the data replication described here is for read-only files. Data replication for read-write enabled files would require an additional discussion concerning distributed update distribution, distributed replica consistency, etc.

In order for a data placement service to support data replication, file name virtualization is needed, e.g., the mapping of a system wide unique identifier to the physical locations of the file. The identifier and the physical locations are already defined in previous sections of this thesis and are called LFN and PFN respectively. Another component that simplifies data replication is a reliable file transfer service that ensures that replicas are valid copies of the original file. In addition to these technical aspects, a data replication solution also requires policies for replication. Policy aspects include, e.g., the selection of replication sources, the selection of replication targets, and the number of replicas that should be created. Furthermore, rudimentary service level agreements can be supported by such policies, e.g., guaranteeing a minimum or maximum number of replicas of a file in the Grid.

The component performing data replication is called the replicator. The replicator uses the LFN to retrieve a list of already existing physical copies of the file inside the Grid. Using the storage discovery and allocation methods as described in Section 4.1 and Section 4.2, the replicator discovers and allocates SEs as new replication targets. After the replicator has gathered all information needed to perform replication and decided upon replication sources and replication targets, the transfer of the files is started. The transfer should be performed by a reliable file transfer service to ensure consistent copies of the original file throughout the Grid. The last step performed by the replicator is the registration of the new physical copies of the file to the LFN in a file virtualization system, e.g., the CatalogService of the DPS or the GT4 RLS.

4.4 Data Consistency Control

Data that is stored in the Grid on SEs might get corrupted over time. An SE crash can accidently corrupt files, a hard disk error can erase files and, network failures, perhaps the most common type of error, can corrupt data transfers. If users are to trust the Grid as a reliable data storage and computation environment, data consistency must be guaranteed. Jobs running inside the Grid also rely on correct input data.

A way to efficiently ensure data consistency is to use checksums generated by a hash function. A hash function is a mathematical function that maps some kind of data, for example the content of a file, to a much smaller, yet unique, hash value often called the checksum. A hash collision means that the hash function maps different data to the same value. Unless the content of the file itself is used as hash value, hash collisions are possible. The problem with a not collision-free hash function is that the same checksum may be mapped to two files with different content. Although it is unlikely, the files on a SE could be replaced by different files and the recalculation of the checksums would generate the same values even though the content of the files has changed. The
probability for such a collision can be minimized by carefully choosing the hash function. In a data Grid, the checksum and the name of the used hash function used can be stored in a central index along with the LFN-PFN mappings. This information can then be used to perform data consistency controls on both the client-side and the SE.

Data consistency is ensured as follows. A program on the SE calculates the checksums of all files on a storage and compares the calculated checksums with the checksums in the central index. If the checksums do not match, possible error handling actions can be performed. The file can be deleted from the SE and its PFN removed from the central index. Another possibility is to replace the corrupted copy of the file with a valid one. In case the only available copy is the corrupted file, its LFN and PFN are removed from the Grid completely. The behavior used can be implemented as a policy for the SEs on an individual SE basis. Furthermore, by matching the list of actual existing files and the list of files posted by the central index, catalog inconsistencies can be resolved. Each SE is thus responsible for the correctness of its entries in the index and the correctness of its own files. This ensures a completely self-checking, self-healing catalog-storage system. However, such a solution may not be feasible and/or scalable due to the tight synchronization requirements.

4.5 Multi-Protocol Reliable File Transfer

Transferring files from a client to the Grid and inside the Grid are common tasks for a data placement service. The correctness of data placement systems and of jobs using input data provided by the Grid relies on correct file transfers. In the latter case, jobs can fail, produce incorrect output, or behave in an unforeseen fashion. As a result, file transfers must be done in a reliable manner. In order to be as robust as possible for clients and the underlying Grid infrastructure, file transfers should be reliable regardless of the transfer protocol used.

Figure 4.4 illustrates reliable file transfer in the general case. The File Transfer Controller is a service that monitors the transfer between A and B. A and B are data locations and represent either a client or an SE. As illustrated with an arrow between the two data locations, a transfer from A to B is to be monitored. B indicates how many bytes of the file it has already received to the File Transfer Controller and whether the transfer is still active. Possible transfer errors include network errors and system errors, the latter including hard disk errors, temporary system overload, power outage etc. The File Transfer Controller is configured to solve possible error situations according to a defined policy. This policy can include, e.g., the maximum number of retries for a retransmission, whether a file should be retransferred from the beginning or from a checkpoint, and, if the transfer consists of a set of files, whether the whole transfer should be aborted if one transfer fails, e.g., removal of already transferred files from the
4.5. Multi-Protocol Reliable File Transfer

Figure 4.5: Multi-protocol reliable file transfer using a transfer cache.

destination. The applicability of such policies is, however, dependent on the transfer protocol used. Transfer protocols such as GridFTP support checkpointing and retransmission, but the presented approach also works with other transfer protocols. There are error situations that cannot be recovered, e.g., if A or B become unavailable, if the File Transfer Controller itself fails or the communication between the data location to be monitored and the File Transfer Controller fails. The latter two cases can be prevented by carefully choosing the placement of the File Transfer Controller. The transfer state can be stored in a data base and thus enable file transfer recovery if the File Transfer Controller crashes.

Most contemporary Grid infrastructures support and use the de-facto standard GridFTP as transfer protocol. Some Grid infrastructures include legacy data servers and data management systems. These systems often have their own transfer protocols. To successfully integrate these systems in the Grid data infrastructure, the file transfer between these legacy systems and the other SEs must be reliable as well. One approach to this problem is to deploy a GridFTP server on the respective data servers and enable them to communicate with the Grid. Another approach is to implement multi-protocol reliable file transfer. Figure 4.5 shows a simple multi-protocol enabled reliable file transfer architecture. Similar to Figure 4.4, the central component in this architecture is the File Transfer Controller. The main difference from Figure 4.4 is, that A and B use different transfer protocols and that a new component, the Transfer Cache is introduced. The Transfer Cache is a special data location that supports several transfer protocols and provides sufficient storage capacity to relay all data that is sent from A to B. All data transferred from A is cached by the Transfer Cache and forwarded to B. The obvious drawback of this approach is that the amount of data transferred is doubled. This can be prevented using the adapter design pattern[22], e.g., the Transfer Cache is placed on A’s or B’s side, acting as an adapter. Existing solutions such as the Stork[27] system use a similar approach to implement multi-protocol reliable file transfer.
4.6 Job-Aware Data Placement Optimization

A job executed in the Grid needs its input data in order to run. The input data might be stored at a different physical location than the machine where the job is to be run. Thus, the input data has to be transferred to the computation node running the job before the job can be executed. If the network between the computation node and the input data is slow, the job is delayed or may even fail if the data arrives too late. There is, however, no easy solution available for this problem.

Job-aware data placement optimization requires interaction between the job scheduler and the data placement service. While the job scheduler is responsible for choosing a suitable computation node, the data placement service is responsible for placing the data on appropriate SEs. The decisions of both components must be combined in order to enable optimal job and data placement. Based on predefined heuristics, good placement decisions can be made. Example heuristics include, e.g., to minimize the amount of transferred data or the transfer time. Furthermore, such heuristics might determine that the latency of the network between data storage and computation node should lie below a certain value, e.g., quality-of-service parameters.

Assuming every computation node has its own file server daemon running, e.g., a GridFTP server, a data placement optimization system can be realized as follows. If a job is sent to a computation node, the computation node uses the data placement service to download a copy of the input file to its local storage. A potentially close SE is chosen by the data placement service before the download process is started. The replication functionality of the data placement service can be used to perform the transfer, thus increasing the availability of the file in the Grid. Furthermore, when the computation node finishes its computation and produced its output files, it uses the data placement service to upload the file to the Grid. Afterwards, the computation node frees its local storage resources by the deletion of both input and output files. Input and output data still exists in the Grid for further computation and analysis, but is of no interest for the computation node after job execution. This is, in fact, the way that the GT4 Grid Resource Allocation and Management (GRAM) interface performs file staging on job execution. GRAM is a set of services to submit, monitor, and cancel jobs on Grid computing resources. Even though GRAM is not a scheduler itself, it offers a job submission interface to job schedulers. GRAM uses the RFT service for reliable file staging.

Thus far, this discussion had made no assumptions about the Grid infrastructure and deployment, e.g., computation nodes and SEs are not arranged in a specific hierarchy. Even though this might represent the common infrastructure, an overlay hierarchy grouping computation nodes and SEs might improve and support job and data placement decisions. In such a setting the computation nodes of a Grid are arranged as computation groups, e.g., computations node on the same subnet form a computation group. These computation groups do not have to be fixed and might rearrange themselves into new dynamic groups. In order to provide a data storage backbone for each computation group, at least one SE is assigned to each group. These SEs are typically connected through a high bandwidth network, typically located on the same subnet and acting as a data cache for the computation groups. The data caches can also be grouped to form a high speed distribution network. Placement decision can take into account the number of computation nodes and data caches that are associated with a computation group. The replication mechanism of the data placement service is used to replicate the necessary input data for the job to the data cache node that acts as
4.6. Job-Aware Data Placement Optimization

Figure 4.6: Job-aware data placement optimization as proposed in [29].

Ranganathan et al. [29] investigate through simulation studies, how job and data scheduling, and data replication can be combined for job and data placement optimization. Their work assumes that data is placed in the Grid first, and the job placement decision is done thereafter. The proposed, simulated architecture, illustrated in Figure 4.6, has three distinct modules, the External Scheduler, the Local Scheduler and the Dataset Scheduler. A user submits a job to the External Scheduler. Based on external information, e.g., load at a remote site or the location of a dataset, the External Scheduler selects a Local Scheduler running at a specific site that is thereafter responsible for the scheduling and execution of the job at this site. Each site has a Dataset Scheduler that keeps track of the popularity of data sets. Based on policies, data sets are replicated to other sites. Ranganathan et al. conclude that job-aware data placement optimization can only be achieved if both placement decisions are combined. The best job placement algorithm identified in the article places a job at the site that contains the data needed by the job, thus minimizing network bandwidth usage. The article further concludes that it does not matter what replication algorithm is used for data replication as long as replication is performed. The presented replication algorithms replicate popular data sets to other sites if a sites computational load exceeds a threshold. However, the article makes no statement about where to place data in the Grid initially.
4.7 Transactions

The operations of a data placement service often involve interaction between several services and hosts. In the DPS, the upload of a file to the Grid involves storage discovery and allocation through the StorageBrokerService, file upload through the file transfer service, and at the end the registration of the file in the system wide CatalogService. It is important for both the DPS itself and the clients of the system that the whole file upload process is performed in a consistent manner. In case the process cannot be performed without a failure, e.g., the client crashes or the CatalogService is temporary unavailable, the system should still remain consistent. Consistency in this context means that if the process cannot be completed, it should either be repeated or the previous system state should be recovered. This illustrates the need for distributed transactions in a data placement service.

In general, distributed transactions describe a bundle of operations that involves two or more participating hosts. A transaction is created and managed by a transaction manager. The manager is responsible for the coordination of the actions that are specified in the transaction. The transaction must have certain properties, typically denoted ACID. ACID is an abbreviation for atomicity, consistency, isolation and durability. Atomicity is the ability of the transaction manager to perform either all of the tasks of the transaction or none of them. If one part of the transaction fails, the whole transaction fails. The consistency property ensures that the system remains in a consistent state both before and after the transaction. In case a part of the transaction fails, the entire transaction is rolled back to restore the initial system state. Isolation means that other operations cannot access or see data in an intermediate state during a transaction. Concurrent transactions are thus safe from interfering with each other through isolation. Durability is a guarantee that once the transaction completes, it is persistent and cannot be undone. By doing so, the transaction can handle system failures.

The ACID properties are of most interest for read-write transactions, e.g., data is read and written during the transaction. The data inside a Grid is mostly handled as read-only data, e.g., on replication. The metadata is, however, read-write data, e.g., the mapping between LFN and PFNs and the file checksum. This data may become inconsistent during a system failure and thus, the ACID theory works well for this scenario.

Distributed transactions are of interest for many Grid applications and services in addition to data placement services. There are proprietary software products that support distributed transactions. The OASIS standardization body defines transactions for WS in three standards: WS-Coordination[5], WS-AtomicTransaction[5], and WS-BusinessActivity[5]. However, contemporary Grid middleware, e.g., the GT4 have not implemented these standards yet.
Chapter 5

Implementation

Based on the ideas from the last chapter a software prototype that resolves many of the discussed issues is implemented. The code base is taken from the Data Placement Service (DPS) reference implementation described in Section 3.5. The goal is to investigate the feasibility of the ideas discussed in Chapter 4. The implementation is done in a change-one-component-a-time fashion in order to avoid breaking the functionality of the existing code. The interfaces described in this chapter are listed in Appendix A for reference.

5.1 Data Replication

By default, the DPS stores one copy of a file in the Grid. Having a single copy in the Grid can lead to problems if an SE becomes unavailable. To overcome this situation, data replication is used. The DPS offered a simple version of data replication in the previous version. All files uploaded to the Grid were stored on two SEs. While this implementation provided basic replication, its biggest drawback was that replication only was supported on file upload as this was the only operation available in the interface. To overcome this, the StorageBroker interface is extended with a method to replicate files anytime, not only as part of upload. This interface defines a replicate operation with the LFN and number of replicas as input parameters. The method returns the number of successful replicas. The semantics of this operation is such that it places at most one copy of the file on each SE. A situation where a file is stored twice on the same SE is thus avoided, as it would improve neither fault tolerance nor performance. Replication is realized as an external component that implements the Replicator interface.

The DPS has two working implementations of the Replicator interface, the SimpleReplicator and the AdvancedReplicator. The difference between the above mentioned SimpleReplicator and the AdvancedReplicator lies in the replicator’s replicate() method implementation, e.g., the way the replicator copies the files from the sources to the sinks. The SimpleReplicator uses single file transfer while the AdvancedReplicator uses bulk file transfer. The advantage of latter is that with the usage of bulk transfers, the overall replication process performance is improved. The bundling allows the transfers to be performed in parallel. This needs not to be true if the source network interface is a bottleneck. The DPS uses the AdvancedReplicator by default.

Figure 5.1 shows the sequence diagram for data replication in the DPS using the AdvancedReplicator. At first, the client calls the replicate() method of the StorageBroker. The StorageBroker forwards the invocation to the replicator, in this case the
AdvancedReplicator. Before the actual file replication can take place, the replicator queries the CatalogService for replication sources, e.g., valid PFNs of the file that is to be replicated, it next queries the StorageBrokerService for new replication sinks, e.g., physical locations where the file can be stored. The list of possible replication sinks has to be matched against the list of replication sources to avoid storing more than one physical copy of a file on the same SE. This matching is necessary as the StorageBrokerService does not offer the possibility to define a filter that specifies to-be-excluded storages. With the filtered list of replication sinks, the file copy process can be initiated. The `replicate(sources, sinks, numReplicas)` method tries to copy the file to at most `numReplicas` replication sinks. In post-processing, the list of old PFNs and new PFNs is merged to a single list and registered in the CatalogService. The replicator then returns the number of successful replications to the StorageBroker, which, in turn, returns this value to the client.

5.2 Data Consistency Control

In Section 4.4, checksums are presented as a solution to the data consistency problem. The old reference implementation’s infrastructure already supported this functionality but did not expose it to the DPS users. The file checksum and the name of the hash algorithm used to calculate the checksum are stored in the CatalogService. Data consistency checks in the old and the new prototype work similarly.

First, the checksum of the original file is calculated and stored with the name of the hash algorithm in the CatalogService. This is done on file upload, as seen in Figure 3.12. Subsequent consistency controls use this checksum for validation purposes. The DPS supports the MD5 and SHA-1 hash algorithms. Furthermore, the StorageBroker interface offers the possibility of performing client-side data consistency controls. This means that the DPS can check whether a local file is a valid copy of a file, the latter identified
5.3 Dynamic Storage Discovery

Dynamic storage discovery is implemented using a central SE index based on the second alternative discussed in Section 4.1. As the GT4 has a flexible index service, the MDS, this has been used for the implementation. In the envisioned scenario, every SE has a MDS running locally, henceforth denoted as the L-MDS. In the Grid one additional MDS is running, called Virtual Organization MDS (VO-MDS). The names are chosen for easier illustration and are by no means special versions of the MDS, e.g., if the computer running the VO-MDS should be an SE as well, the L-MDS and the VO-MDS are the same. The L-MDSs are configured to stream their information upwards to the VO-MDS, as shown in Figure 5.3. The information registered in the MDSs has a limited lifetime and thus expires after a pre-configured time. The SEs therefore have to refresh their registrations periodically. If an SE becomes unavailable and cannot refresh its registration, its entry is automatically deleted from the VO-MDS. By doing so, the VO-MDS gets an updated view of what SEs are available. The dynamic storage discovery consists of two parts, the SE publication and the actual SE discovery.

SE publication is illustrated in Figure 5.3. A shell script, called storage_element.sh in the figure, collects information about the SE, e.g., the host name, the type of service, and stores it in the L-MDS. The L-MDS then streams this information to the VO-MDS, which maintains a central index of all available SEs.

Figure 5.2: Sequence diagram for client-side data consistency check.

by its logical file name. Figure 5.2 shows the sequence diagram for the client-side data consistency check with a SHA-1 checksum. Two DPS components are needed for the consistency check, the CatalogService and the hash generator. In this scenario, the CatalogService is queried for the file checksum and the name of the hash algorithm. The appropriate hash generator, SHA1Hash, is then selected by HashFactory and invoked with the local file as argument. The two checksums are then compared and the result is reported back to the user.
(GridFTP or FTP) it is running, the current system time, and file system information. This information is rendered to an XML document. The shell script is used as Execution Source, that in turn aggregates the information to the L-MDS using the Aggregator Framework. The information stored in the L-MDS is next propagated up to the VO-MDS. The SE discovery part is implemented as WSRF resource property queries to the MDS.

### 5.4 Reliable File Transfer

The DPS performs intra-Grid transfers with file replication and transfers between clients and the Grid with file upload and download. To increase reliability of the DPS, two file transfer components are implemented, one for intra-Grid transfers and one for transfers between client and Grid.

The RFT service supports reliable third-party file transfer and is therefore used by the DPS for intra-Grid file transfers. GT4 provides a command-line interface for RFT, but no Java API. Using the GT4 client’s Java code-base, a simple Java RFT client has been implemented. In contrast, for client-Grid transfers, the client had to be implemented from scratch. This client implements a finite-state machine (FSM) with five possible states, pending, active, done, retry and fail. Figure 5.4 shows the FSM with the states and the transitions allowed between them. A transfer is represented by a Transfer object. This object holds the transfer state, the transfer type, as well as the transfer source and the transfer sink. Possible transfer types are client to server, server to server, and server to client. GT4 provides a Java GridFTP client that can be used to communicate with GridFTP and FTP servers. The client leverages on various features and configuration options in the GridFTP, in particular, the transfer markers that carries information about the current transfer. There are two classes of markers: performance markers and restart markers. The restart markers report how many bytes of the file that already have been written (or read) on the server side. This information
Figure 5.4: Finite-state machine for reliable file transfer.

Figure 5.5: Sequence diagram of the SE handler dispatcher.

can be used to restart the failed transfer from where it halted. The performance markers are of no interest for the implementation of the reliable file transfer client.

When a transfer is created, its initial state is pending. As soon as the transfer starts, its state changes to active and the client is caching the restart markers sent by the server. If an error occurs during file transfer, the transfer’s state changes to fail. If the internal counter for failed transfer attempts exceeds a predefined, but configurable, value, max-retry, the client exists gracefully and reports the reason for the transfer failure. If the counter is below max-retry, the transfer’s state is changed from fail to retry. In this state, the restart marker cache is read and is used to configure the transfer options of the internally used Java GridFTP client to restart from where the transfer stopped. If the transfer completes successful, its state is changed to done. The reliable file transfer functionality is provided in a transparent manner for the StorageBroker. To interface SEs, the DPS utilizes the adapter design pattern[22] through the StorageElementHandler interface.

This interface defines methods for simple SE interaction, e.g., file upload, file download, file deletion, file and size queries. An SE handler dispatcher selects the appropriate implementation of the interface on method invocation. This mechanism enables support for other transfer protocols to be added through simple implementation of the StorageElementHandler interface and registration of the new handler in the dispatcher. The
DPS has three SE handlers for GridFTP storages: GridFTPHandler, ExtendedGridFTPHandler and ReliableFileTransferHandler. The GridFTPHandler is the simplest version and executes unreliable file transfer. The ExtendedGridFTPHandler extends the GridFTPHandler with reliable client-server and server-server file transfer as described above. Finally, the ReliableFileTransferHandler extends the ExtendedGridFTPHandler and replaces the reliable server-server file transfer method by interfacing with RFT. The DPS uses the ReliableFileTransferHandler by default, e.g., client-Grid transfers are performed using ExtendedGridFTPHandler’s functionality, while intra-Grid transfers are handled by the RFT. This approach is used as RFT only supports intra-Grid transfers.

Figure 5.5 shows the sequence diagram of a server to server file transfer. In this scenario, the source and sink parameters are GridFTP URLs. The StorageElementHandlerDispatcher thus selects the ReliableFileTransferHandler as SE handler. The ReliableFileTransferHandler calls the RFTServiceConnector to execute the file transfer using the RFT service. The RFTServiceConnector, shown in the figure, is the simple Java RFT client described above.
Chapter 6

Discussion

In this chapter we discuss the benefits and limitations of the DPS. To broaden the horizon, related work in the field of data management and data placement for Grid computing is surveyed. This chapter is concluded with ideas for future work based on the capabilities of the current system.

In this thesis, data placement issues are discussed and those that are selected are implemented in the DPS. Command line tools enable easy user interaction with the system alongside with the reworked Java client API. File transfers between clients and Grids are performed in a reliable manner adding robustness to the file transfer subsystem of the DPS. Dynamic storage discovery is possible through the utilization of the MDS. The system is, however, not ready for production use yet. No fallback scheme implemented for it yet. Component failure, especially on the client-side, can leave the system in an undefined state, rendering the results of later operations undefined. Furthermore, the installation and configuration is complicated. The DPS has been tested on a small-scale Grid with four participating computers acting as both SE and DPS, i.e., the DPS was accessed through different computers and users. The DPS is in a development stage, ready to be used for validation and further prototyping.

6.1 Related Work

This section briefly discusses related work in the data placement area and compares the DPS reference implementation to existing solutions.

The Smart Storage Element (SSE) is developed by the NorduGrid project as part of the Advanced Resource Connector (ARC) software[26]. The SSE is part of ARC’s Data Storage Infrastructure (DSI). The DSI consists of SSEs, Data Index Services (Data IS), and the Indexing Services Infrastructure (ISI). The Data IS is interfacing the ISI. Currently, the Globus Replication Catalog (RC) and RLS are used as indexing service back ends. The SSE itself is integrated in the HTTPSD server[25] and is associated with at least one IS. Thus the SSE handles registration of the uploaded files in a manner transparent to the user. If a user requests a file from an SSE and the file is not located on the SSE, the SSE contacts neighboring SSEs to obtain a copy of the file. The same mechanism is used to replicate files within the DSI. HTTPS and HTTPG are used as data transfer protocols. The SSE is also accessible through the SRM interface. The DPS developed as part of this thesis and the ARC SSE share some similarities. The basic concept of a metadata catalog exists in both systems. In our design the
metadata catalog resides in the CatalogService. In ARC’s Data Storage Infrastructure, the metadata catalog resides in the Data IS and ISI, whereas the SSE operates almost entirely free from human intervention; more manual work is needed when using the DPS. This is due to the already mentioned integration of the SSE in the HTTPS server. By contrast, the DPS approach represents a more open and loosely-coupled architecture.

The dCache data management system is a joint venture between the Deutsches Elektronen-Synchrotron (DESY) and the Fermi National Accelerator Laboratory (FNAL). One of the key design concepts of dCache is that the location and multiplicity of the data is autonomously determined by the system based on configuration, CPU load, and disk space. The dCache name space is uniquely represented within a single file system tree. In dCache, storage elements are called pools. These pools can be added and removed at any time. Data availability is guaranteed by replication over several disc pools. dCache is thus tolerant against failures of pools. The pools are presented to the user as one federated hard drive. The system automatically load balances the data across the pools to decrease access times to the files. File access is supported through a native protocol called dCap that can be used via command line tools or a C client API. It is also possible to access dCache through GridFTP. Other protocols can be implemented as needed through a provided interface. Furthermore, dCache supports the SRM interface, enabling it to be used in generic data Grid environments like the LHC Grid. Although both handle data placement, the respective goals of dCache and the DPS differ. The level of abstraction that dCache uses for storages is higher than the one in DPS. The dCache system is designed to work with tertiary storage managers and intelligent dynamic data staging. Furthermore, dCache takes the configuration, CPU load and disk space of the storages into account when replicating files, whereas DPS only considers the storage disk space.

The Storage Resource Broker (SRB) is a data management system developed at the San Diego Supercomputer Center (SDSC). It provides global persistent identifiers for naming files across several storages. The network spread storages are presented as one global file system. A large variety of storage types are supported. SRB supports replication of files between sites and caching of copies of files in local storage systems. To enforce consistency constraints, SRB makes use of transactions. Similar to the DPS, the central part of the SRB system design is the metadata catalog. Unlike the DPS, the SRB supports the creation of collections and thus organizes data in a different way in the catalog. SRB also supports federation of servers, which is not possible with DPS.

Researchers at the university of Wisconsin-Madison developed the Stork data placement scheduler. The Stork system handles data placement in analogously to the way computational jobs are managed, in other words, data placement jobs are queued, scheduled, monitored, managed and even checkpointed. The system is designed to achieve maximum reliability with as little human intervention as possible. An unusual feature of the Stork system is the support for protocol translation, e.g., the system is able to translate transfer protocol messages between a variety of transfer protocols such as FTP, GridFTP, SRB, SRM and UniTree. This is achieved with a mechanism similar to the Transfer Cache described in Section 4.5. Furthermore, Stork supports run-time protocol auto-tuning to achieve optimal bandwidth usage. Each storage element in the Stork system can offer different protocols. Stork selects the best-fitting protocol while taking into account dynamic environmental changes such as network congestion, link failures, and even file system errors. The aims of the DPS and Stork are similar, as both projects focus on data placement. Scheduling data placement tasks such as computational jobs separates Stork both from DPS and from all the other projects presented here.
The Laser Interferometer Gravitational Wave Observatory (LIGO) is a multi-site national research facility whose objective is the detection of gravitational waves\cite{3}. The LIGO project uses a system called Lightweight Data Replicator (LDR)\cite{13} to provide end-to-end management of collected data. The LDR is a high-level data management system. It is responsible for initiating and monitoring data replication. Files are represented by logical file names. Like the DPS, LDR uses Globus RLS as catalog service and GridFTP as transfer protocol. The LDR performs data consistency control as described in Section 4.4 on SE level using Python scripts.

### 6.2 Future Work

Further enhancements of the DPS could focus on improving the robustness of the current implementation as well as simplifying installation and configuration. System robustness could be achieved through the implementation of a transaction mechanism. If each process (upload, replication etc.) of the system is modeled as a transaction, the system remains in a consistent state even if failures occur. WS-Transaction\cite{5} seems to be a possible candidate to realize this functionality in the DPS.

The currently used GIRD Grid job management software is only aware of GridFTP URLs and thus the DPS uses early binding. Early binding in this context means that a PFN is included in the job description before the job is submitted. As the relationship between the LFN and PFN is resolved before the job is submitted, this binding is called early. Conversely, late binding in this context means that the LFN is included in the job description before submission of the job. The resolving from LFN to PFN is performed on the initial access, or even on every access to the file. The motivation to integrate support for late binding in the job management tool is the loose coupling achieved between a job and its physically stored data, which improves robustness and fault tolerance, e.g., against failing SEs. The job only has to keep track of the LFN(s) and receives a valid copy of the file on demand. Support for both late and early binding is already implemented in the DPS. The drawback of late binding is that job management tools have to be modified to be catalog-aware in order to resolve LFNs.

In addition to these functionality improvements, possible future directions for this work include a performance evaluation to investigate the scalability of the DPS in terms of users and SEs.
I would like to thank all of the following persons who helped me with this thesis project. To Johan Tordsson, my supervisor at the Department of Computing Science, thank you for the help with report writing and inspiring discussions. To Henrik Thostrup Jensen, thank you for all the insightful suggestions during the investigation of related work. Thanks also goes to P-O Östberg for his help dealing with the wonderful world of Grid security. I also acknowledge Tomas Ögren for providing me with technical support and a computer cluster that I could freely use for arbitrary development and prototyping. To Mattias Wadenstein from the High Performance Computing Center North (HPC2N) [4], thank you for the insights in what real-world data center installations look like.

Furthermore, I would like to thank my girlfriend Ulrika Back for supporting me throughout the entire thesis. To my study colleagues and friends Raphaela Bieber, Sebastian Bardt, Michael Brandl and Nikolay Georgiev, thank you for the moral support. Special thanks go to Sebastian Bardt for countless discussions and coffee breaks - they have added a lot value to the thesis. Additional special thanks go to Timothy Barbitta for proof-reading the final version of the thesis.

Last but not least, I especially thank my parents for enabling me to study abroad in Umeå. Thank you for giving me the opportunity to broaden my horizons.
References


[22] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Longman, Amsterdam, 1995.


Appendix A

DPS Interfaces

This appendix chapter contains interfaces mentioned in the thesis. Each interface is briefly described. These interfaces are not guaranteed to be stable in upcoming releases.

A.1 StorageBroker Interface

The StorageBroker interface represents the client side public interface to the DPS. It defines the possible interaction between clients and the DPS. Interaction with the DPS should only be performed through this interface.

Listing A.1: The StorageBroker interface

```java
package se.grid.storage;
import java.io.File;
import java.io.IOException;
import org.ietf.jgss.GSSException;

public interface StorageBroker {
    public String uploadToGrid(File f, HashType hashAlgorithm)
        throws IOException, GSSException;
    public void downloadToFile(String lfn, File f, boolean consistencyCheck)
        throws IOException, GSSException;
    public void deleteFromGrid(String lfn) throws IOException, GSSException;
    public boolean checkDataConsistency(String lfn, File f)
        throws IOException, GSSException;
    public int replicate(String lfn, int numReplicates)
        throws IOException, GSSException;
}
```
A.2 Replicator Interface

The Replicator interface defines a method for data replication. It has to be implemented by all replication algorithms used inside DPS.

Listing A.2: The Replicator interface

```java
package se.grid.storage;
import java.io.IOException;
import org.ietf.jgss.GSSException;
public interface Replicator {
    public int replicate(String lf, int numReplicates) throws IOException, GSSException;
}
```

A.3 StorageElementHandler Interface

The StorageElementHandler interface defines methods for simple SE interaction. These actions include file upload, file download, file deletion, file size queries, one-to-many and many-to-many file transfers. Every SE type that is to be supported by DPS must implement this interface.

Listing A.3: The StorageElementHandler interface

```java
package se.grid.storage;
import java.io.File;
public interface StorageElementHandler {
    public boolean put(File f, String uri);
    public boolean get(String uri, File f);
    public boolean delete(String uri);
    public void delete(String[] uris);
    public long size(String uri);
    public boolean exists(String uri);
    public boolean put(String source, String sink);
    public void put(String source, String[] sinks);
    public void put(String[] sources, String[] sinks);
}
```
Appendix B

Installing the Data Placement Service

This chapter gives a brief installation guide to the Data Placement Service. The purpose of this appendix is to provide interested readers with the possibility of trying the Data Placement Service.

B.1 Requirements

Prior to the installation of the DPS, the GT4 has to be installed and configured on all participating computers, e.g., both clients and storage elements. The current version of the DPS has been developed and tested with GT4 4.0.7 under Ubuntu 8.04. The DPS requires that GT4 is configured and runs without problems. Even though DPS does not utilize all GT4 components, a subset of the GT4 components has to be configured properly. Components that are used by the DPS include the RFT service, the MDS and the RLS.

B.2 Download Software

The DPS development release is split into three archives to simplify distribution, development and maintenance. All archives can be downloaded from:

http://www.cs.umu.se/~ens07mhg/edu/dps/

B.3 Configure the DPS

To configure and install the software, the downloaded archives must first be unpacked to a directory, e.g. to /tmp/dps/. The following configuration instructions assume that the archives are unpacked to /tmp/dps/, e.g., all files concerning the CatalogService are stored under /tmp/dps/CatalogService/, all files of the StorageBrokerService are stored under /tmp/dps/StorageBrokerService/, and all files from the storage element archive are stored under /tmp/dps/storage-element/. Furthermore, the instructions assume that the GT4 is installed under /opt/globus/. The DPS services have to be configured before the software can be installed in the GT4 Java WS core.
Enable logging. GT4 and DPS use log4j for logging purposes. Enabling logging for
the DPS is recommended, as the system is still in proof-of-concept state. Furthermore,
the feedback from the system can be used in upcoming development cycles and sys-
tem improvements. Logging for all Java WS in GT4 is configured through the central
container-log4j.properties file, which is located in the GT4 installation directory.

The following lines have to be added to this file to enable logging for the DPS:

log4j.category.se.grid.storage.catalog=DEBUG
log4j.category.se.grid.storage.broker=DEBUG

CatalogService configuration. The CatalogService comes with two back ends, a
JDBC-driven back end and a RLS-driven back end. Even though the JDBC back end
has been tested successfully, its usage is deprecated as the focus of the latest version of
the DPS was on RLS integration. The back end that should be used by the CatalogSer-
vie is selected through the server-config.wsdd web service deployment description
that is found in the directory of the CatalogService. The selected back end has to be
uncommented in the file. Conversely, the selected back end that is not selected must be
disabled as a comment in the file. The corresponding part of the file looks like this:

[...]
<!-- WS implementations -->
<parameter
   name="className"
   value="se.grid.storage.catalog.impl.JDBCCatalogService"/>
<parameter
   name="className"
   value="se.grid.storage.catalog.impl.RLSCatalogService"/>
[...]

The configuration of the used back end is stored in config.xml under etc/ of the
CatalogService directory. If the RLS back end is used, only the RLS server URL must
be specified. Otherwise, the database configuration must also be configured. This
configuration file is self-descriptive. A sample config.xml file for the CatalogService
looks like this, assuming the RLS runs on localhost:

<?xml version="1.0" encoding="UTF-8"?>
<ns1:engineConfigType xmlns:ns1="http://grid.se/storage/catalog"
   <!-- Configuration of the RLSCatalogService backend -->
   <ns1:rls>rls://localhost</ns1:rls>
</ns1:engineConfigType>

StorageBrokerService configuration. The configuration of the StorageBrokerSer-
vie itself is easy. Only the storage discoverer and the storage selector have to be
adjusted. Both settings are configurable in the config.xml file which is located under
etc/ inside the StorageBrokerService directory. An example configuration, assuming
that dynamic storage discovery is used an the VO-MDS is running on vo-mds looks as
follows:

<?xml version="1.0" encoding="UTF-8"?>
<ns1:engineConfigType xmlns:ns1="http://grid.se/storage/broker">
B.3. Configure the DPS

```xml
<ns1:storageDiscovery>
  se.grid.storage.broker.impl.MDSStorageDiscovery
</ns1:storageDiscovery>

<ns1:indexService>
  https://vo-mds:8443/wsrf/services/DefaultIndexService
</ns1:indexService>

<ns1:storageSelection>
  se.grid.storage.broker.impl.WorstFitStorageSelection
</ns1:storageSelection>
</ns1:engineConfigType>
```

As the current code base for both the StorageBrokerService and the StorageBroker is mixed, the configuration for the StorageBroker is hidden within the source code. This is an inflexible solution and should be changed in future releases of the DPS. The settings for the StorageBroker are located in `Configuration.java` under the path `src/se/grid/storage/impl/` in the StorageBrokerService directory. The following addresses have to be set correctly to guarantee that the StorageBroker can work properly:

- The address of the Reliable File Transfer factory service.
- The address of the Reliable File Transfer service.
- The address of the StorageBrokerService.
- The address of the CatalogService.

These values are not set to `localhost` and should be specified as full-qualified domain names instead of `localhost` as the latter can lead to problems if the security infrastructure is not configured properly.

**Storage element configuration.** As described earlier, the storage discovery mechanism used in the DPS is based on GT4 MDS and the underlying Aggregator framework. Each storage element must have MDS configured and running. In general, the following tasks have to be performed in order to enable dynamic storage discovery:

1. Install the aggregator source script in the correct directory (`libexec/aggrexec/`).
2. Configure MDS to accept the aggregator source script as a valid aggregator source.
3. Create a registration file for the aggregator source.
4. Configure MDS to aggregate information to the VO-MDS.

The `storage_element.sh` shell script from the storage element archive¹ has to be copied to the `libexec/aggrexec/` directory of the GT4 installation on each computer that should be used as a storage element. In addition, the shell script has to be configured as valid aggregation source in the MDS configuration file `jndi-config.xml`, which is located under `etc/globus_wsrft_mds_index/` of the GT4 installation.

The corresponding section of this file looks as follows:

¹One of the downloadable archives is called storage element archive.
The storage element archive comes with an example registration file that has to be edited for the corresponding environment. A sample registration file is shown below, assuming the storage element’s name is `test-storage-element`:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<ServiceGroupRegistrations
 xmlns="http://mds.globus.org/servicegroup/client"
 xmlns:sgc="http://mds.globus.org/servicegroup/client"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing"
 xmlns:agg="http://mds.globus.org/aggregator/types">
  <defaultServiceGroupEPR>
    <wsa:Address>
      https://test-storage-element:8443/wsrf/services/DefaultIndexService
    </wsa:Address>
  </defaultServiceGroupEPR>

  <!-- The defaultRegistrantEPR defines the grid resource that will be
       registered, unless overridden in the ServiceGroupRegistrationParameters
  -->
  <defaultRegistrantEPR>
    <wsa:Address>
      https://test-storage-element:8443/wsrf/services/storageElement
    </wsa:Address>
  </defaultRegistrantEPR>

  <defaultSecurityDecriptorFile>some/other/sec/file.xml</defaultSecurityDecriptorFile>

  <!-- test-storage-element -->
  <ServiceGroupRegistrationParameters
   xmlns="http://mds.globus.org/servicegroup/client">
    <ServiceGroupEPR
     xmlns:agg="http://mds.globus.org/aggregator/types"
     xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">
      <wsa:Address>
        https://test-storage-element:8443/wsrf/services/DefaultIndexService
      </wsa:Address>
    </ServiceGroupEPR>
  </ServiceGroupRegistrationParameters>
</ServiceGroupRegistrations>
```
B.3. Configure the DPS

</ServiceGroupEPR>

<RegistrantEPR
  xmlns:agg="http://mds.globus.org/aggregator/types"
  xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">
  <wsa:Address>
    https://test-storage-element:8443/wsrf/services/storageElement
  </wsa:Address>
</RegistrantEPR>

<!-- Renew this registration every 60 seconds (1 minute) -->
<RefreshIntervalSecs>60</RefreshIntervalSecs>

<Content xsi:type="agg:AggregatorContent"
  xmlns:agg="http://mds.globus.org/aggregator/types">
  <agg:AggregatorConfig xsi:type="agg:AggregatorConfig">
    <agg:ExecutionPollType>
      <!-- Run our script every 30,000 milliseconds (30 seconds) -->
      <agg:PollIntervalMillis>30000</agg:PollIntervalMillis>
      <!-- Specify our mapped ProbeName registered in the
           $GLOBUS_LOCATION/etc/globus_wsrf_mds_index/jndi-config.xml
           file -->
      <agg:ProbeName>storageElement</agg:ProbeName>
    </agg:ExecutionPollType>
  </agg:AggregatorConfig>
  <agg:AggregatorData/>
</Content>
</ServiceGroupRegistrationParameters>
</ServiceGroupRegistrations>

Using this registration file, the mds-servicegroup-add has to be invoked to register
the storage element in the local MDS. It is important to know that the registration
tool must be run in order to refresh the registration of the storage element, e.g., once
configured the mds-servicegroupd-add tool should be run, e.g., as a cron-job or a
background process. The registration is updated periodically based on the settings in
the registration file.

At this point, all storage elements aggregate their information only to the local
MDS. In order to aggregate the information to a centralized MDS, called the VO-
MDS here, a local MDS must be selected as VO-MDS and the other MDSs must be
configured in order to aggregate their information to the VO-MDS. The file that controls
this configuration setting of the MDS is named hierarchy.xml and is located under
etc/globus_wsrf_mds_index/. An example configuration (assuming that vo-mds acts
as the VO-MDS) looks as follows:

<config>
  <upstream>
    https://vo-mds:8443/wsrf/services/DefaultIndexService
  </upstream>
</config>
B.4 Install the DPS

After the services are configured, the DPS is installed by executing the command `ant all` in the respective directories of CatalogService and StorageBrokerService. The latter `ant` invocation is mandatory as some settings are specified in the services and clients. After `ant` finishes without errors two `gar`-archives are generated. These archives contain the properly configured and ready to deploy services. By using the `globus-deploy-gar` tool from GT4 these archives are deployed to the Java WS core.

B.5 Use the DPS

After the `gar`-archives have been deployed to the Java WS core the GT4 container has to be restarted in order to enable usage of the DPS. Upon container startup, the DPS components show some messages, the amount depends on the selected log level. If the container starts without severe warnings, the DPS is ready for use. Due to the security settings of the DPS, a valid user proxy certificate is necessary prior to using the DPS or components of the DPS. The GT4 command line tool `grid-proxy-init` can be used to create such a proxy. The DPS has a command line interface that is briefly described in Appendix C.2. Furthermore, Appendix D can be consulted for example usage of these command line tools.
Appendix C

Public Interface of the Data Placement Service

Currently, the DPS offers two public interfaces, a client API and a command line interface. The client library is designed for application developers who want to include DPS support in their application. The command line interface is provided to give DPS users the possibility to interact with the system directly without the need to implement their own clients.

C.1 Client Library

The client library for the DPS is written in Java and implements the StorageBroker interface. This interface is shown in Appendix A. Functionalities provided by the library are file transfer, file replication and file verification. Application developers can use this library to integrate DPS in their applications, make them DPS-aware, and also to implement future data placement functionalities leveraging on the DPS.

C.2 Command Line Interface

Based on the Java client API library, a command line interface has been implemented as an example application. Basically, these command line interface tools implement file-system style access to the DPS. The tools enable simple operations like file upload, file download, file replication, file verification and file removal. If the tools are executed with either incorrect parameters or none at all, a help text giving usage instructions is displayed. Furthermore, if something goes wrong during execution, the tool exits with an error message describing what happened. A sample usage of the tools is shown in Appendix D.

```
dps-cp-to-grid [HASH_ALGORITHM] FILENAME
```

The `dps-cp-to-grid` tool copies a local file to the Grid. The file specified by `FILENAME` is copied to the Grid, registered in the CatalogService and its logical file name is returned. The parameter `HASH_ALGORITHM` is optional and allows the user to choose the hash algorithm to be used to calculate the checksum.
**dps-cp-to-file** LFN FILENAME [CONSISTENCY CHECK]
The inverse functionality to **dps-cp-to-grid** is provided by **dps-cp-to-file**. This tool copies a file that is stored in the Grid, referenced by **LFN**, to a local file, specified by **FILENAME**. The last parameter is optional and specifies whether the downloaded file should be checked for data consistency. By default the tool checks the consistency of the downloaded file.

**dps-rm** LFN
A file might be stored on several storages inside the Grid. The **dps-rm** tool removes all replicas of the file referenced by **LFN** from the Grid and unregisters the file from the CatalogService. However, it does not unregister the file from the CatalogService if a replica of the file within the Grid cannot be deleted.

**dps-verify-file** LFN FILENAME
The **dps-verify-file** tool enables client-side data consistency checks. The local file **FILENAME** is compared with the file in the Grid, referenced by **LFN**. Even though the verification process is based on file checksums, there is no need to specify a hash algorithm as the system determines the correct algorithm automatically. This tool can also be used to implement storage element side data consistency checks.

**dps-replicate-lfn** LFN NUM_REPLICAS
If a file is to be replicated inside the Grid, the **dps-replicate-lfn** tool provides this functionality. In order to replicate a file, it has to be uploaded to the Grid and its **LFN** has to be known. The **dps-replicate-lfn** tool replicates the file at most **NUM_REPLICAS** times. After execution, the tool returns the number of successfully created replicas, that may be less than **NUM_REPLICAS**, but is always greater than zero.
Appendix D

Sample Command Line Tools Output

This appendix shows example usage of the Data Placement Service command line tools that are described in Section C.2.

D.1 Upload a File

First we upload a file called foobar.txt to the Grid using the dps-cp-to-grid command. For this test, four storage elements roentgen-{1,2,3,4}.cs.umu.se are used. To show more of the system internals, debug output has been enabled.

```
michael@roentgen-4:/tmp$ ls -lha foobar.txt
-rw-r--r-- 1 michael michael 65 2008-09-01 15:23 foobar.txt
michael@roentgen-4:/tmp$ dps-cp-to-grid foobar.txt
```

```
CatalogService : https://roentgen-4.cs.umu.se:8443/wsrf/services/CatalogService
StorageBrokerService: https://roentgen-4.cs.umu.se:8443/wsrf/services/StorageBrokerService
Local file : foobar.txt
```

```
StorageBrokerImpl [main,uploadToGrid:146] 1. Generated UUID
'730e7c70-a29c-11dd-bcef-cf721e03f4f4'
StorageBrokerImpl [main,uploadToGrid:150] 2. Generated file checksum
'10b1f8e6a52a6e7c3ef87b0e0e1c3ef543803'
StorageBrokerImpl [main,uploadToGrid:153] 3. Query StorageBrokerService
StorageBrokerImpl [main,delegateCredentials:116] Connect to StorageBrokerService
StorageBrokerImpl [main,delegateCredentials:132] Delegating user credentials ..
StorageBrokerImpl [main,delegateCredentials:140] DONE
StorageBrokerImpl [main,uploadToGrid:161] Found 4 possible storages
StorageBrokerImpl [main,uploadToGrid:164] 4. Upload file to Grid
ExtendedGridFTPHandler [main,put:282] Put: [PENDING]
'gsiftp://roentgen-2.cs.umu.se:2811//home/michael/759d1690-a29c-11dd-aafe-8e00a8c2b63a'
ExtendedGridFTPHandler [main,reliableTransfer:237] 1. attempt to perform reliable CLIENT_TO_SERVER transfer.
Transfer [main,setState:89] State transition: PENDING -> ACTIVE
MyMarkerListener [main,markerArrived:36] Performance marker arrived 1.224943036E9
MyMarkerListener [main,markerArrived:36] Performance marker arrived 1.2249430361E9
MyMarkerListener [main,markerArrived:32] Current transfer state:
```
D.2 Verify a File

Next, we verify that the original file `foobar.txt` is identical to the file in the Grid using the `dps-verify-file` command.

```
michael@roentgen-4:/tmp$ dps-verify-file 730e7c70-a29c-11dd-bcef-cf721e03f4f4 foobar.txt
```

StorageBrokerImpl [main,checkDataConsistency:277] 1. Look-up in Catalog
StorageBrokerImpl [main,getCatalogServiceInstance:99] Connect to CatalogService
StorageBrokerImpl [main,delegateCredentials:132] Delegating user credentials..
StorageBrokerImpl [main,checkDataConsistency:288] 2. Verify checksum (SHA-1-hash, '10b1f8e6a5526e07c3ef87b5f0ec1accd3efa403')
StorageBrokerImpl [main,checkDataConsistency:291] Verified data consistency: 10b1f8e6a5526e07c3ef87b5f0ec1accd3efa403

The file 'foobar.txt' is a true copy of the file '730e7c70-a29c-11dd-bcef-cf721e03f4f4' in the Grid.

michael@roentgen-4:/tmp$
D.3 Replicate a File

Next, we trigger the replication of our uploaded file using the `dps-replicate-lfn` command.

```
michael@roentgen-4:/tmp$ dps-replicate-lfn 730e7c70-a29c-11dd-bcef-cf721e03f4f4 3
```

```
dps-replicate-lfn DPS client v.1.0.0
CatalogService : https://roentgen-4.cs.umu.se:8443/wsrf/services/CatalogService
StorageBrokerService: https://roentgen-4.cs.umu.se:8443/wsrf/services/StorageBrokerService
Logical File Name : 730e7c70-a29c-11dd-bcef-cf721e03f4f4
Requested replicas : 3
```

1. Look-up in Catalog
2. Check replication source size
3. Query StorageBrokerService for replica sinks
4. Replicate the file in the Grid

```
SimpleReplicator [main,replicate:105] 1. Look-up in Catalog
SimpleReplicator [main,replicate:123] Replication source has 65 bytes.
SimpleReplicator [main,replicate:148] 4. Replicate the file in the Grid
```

```
```

SimpleReplicator [main,matchSourcesAndSinks:74] Replication sink
```
'gsiftp://roentgen-1.cs.umu.se:2811//home/michael/0a3fa010-a29d-11dd-aafe-8e00a8c2b63a' confirmed.
'gsiftp://roentgen-3.cs.umu.se:2811//home/michael/0a56d190-a29d-11dd-aafe-8e00a8c2b63a' confirmed.
'gsiftp://roentgen-4.cs.umu.se:2811//home/michael/0a69bd50-a29d-11dd-aafe-8e00a8c2b63a' confirmed.
```


```
AdvancedReplicator [main,replicate:47] Replicate file
```

RFTServiceConnector [main,transfer:89] Generating complex transfer request.

```
RFTServiceConnector [main,transfer:106] Number of transfers in this request: 3
```

```
```

```
```

```
```

```
AdvancedReplicator [main,replicate:58] Check replicas
```

```
GridFTPHandler [main,exists:195] Exists: 'gsiftp://roentgen-1.cs.umu.se:2811//home/michael/0a3fa010-a29d-11dd-aafe-8e00a8c2b63a'
```

```
```

```
```
Exists: 'gsiftp://roentgen-4.cs.umu.se:2811/home/michael/0a69bd50-a29d-11dd-aafe-8e00a8c2b63a'
AdvancedReplicator [main,replicate:75] Successfully replicated to 3 storages.
SimpleReplicator [main,replicate:151] 5. Register new replicas in CatalogService

The file '730e7c70-a29c-11dd-bcef-cf721e03f4f4' has been replicated 3-times in the Grid.
michael@roentgen-4:/tmp$

We try to replicate the file once again, to ensure that it is already available on all four storages.
michael@roentgen-4:/tmp$ dps-replicate-lfn 730e7c70-a29c-11dd-bcef-cf721e03f4f4 3
dps-replicate-lfn DPS client v.1.0.0
Copyright 2008 Michael Hoefling
CatalogService : https://roentgen-4.cs.umu.se:8443/wsrf/services/CatalogService
StorageBrokerService: https://roentgen-4.cs.umu.se:8443/wsrf/services/StorageBrokerService
Logical File Name : 730e7c70-a29c-11dd-bcef-cf721e03f4f4
Requested replicas : 3

SimpleReplicator [main,replicate:105] 1. Look-up in Catalog
StorageBrokerImpl [main,GetCatalogServiceInstance:99] Connect to CatalogService
StorageBrokerImpl [main,delegateCredentials:132] Delegating user credentials ..
StorageBrokerImpl [main,delegateCredentials:140] DONE
SimpleReplicator [main,replicate:121] 2. Check replication source size
GridFTPHandler [main,size:177]
Size: 'gsiftp://roentgen-1.cs.umu.se:2811//home/michael/0a3fa010-a29d-11dd-aafe-8e00a8c2b63a'
SimpleReplicator [main,replicate:123] Replication source has 65 bytes.
StorageBrokerImpl [main,GetStorageBrokerServiceInstance:116] Connect to StorageBrokerService
StorageBrokerImpl [main,delegateCredentials:132] Delegating user credentials ..
StorageBrokerImpl [main,delegateCredentials:140] DONE
SimpleReplicator [main,replicate:136] 0 storages left after matching.
Something went wrong while replication:
   No storage available in the Grid
michael@roentgen-4:/tmp$
D.4 Download a File

We download the file to the local file `download.txt` using the `dps-cp-to-file` command.

```
michael@roentgen-4:/tmp$ dps-cp-to-file 730e7c70-a29c-11dd-bcef-cf721e03f4f4 download.txt
```

```
dps-cp-to-file DPS client v.1.0.2
```

```
CatalogService : https://roentgen-4.cs.umu.se:8443/wsrf/services/CatalogService
StorageBrokerService: https://roentgen-4.cs.umu.se:8443/wsrf/services/StorageBrokerService
Logical File Name : 730e7c70-a29c-11dd-bcef-cf721e03f4f4
Local file : download.txt
```

```
StorageBrokerImpl [main,downloadToFile:190] 1. Look-up in Catalog
StorageBrokerImpl [main,getConnection:99] Connect to CatalogService
StorageBrokerImpl [main,delegateCredentials:132] Delegating user credentials ..
StorageBrokerImpl [main,delegateCredentials:140] DONE
StorageBrokerImpl [main,downloadToFile:200] 2. Download file
ExtendedGridFTPHandler [main,get:288] Get: [PENDING]
  'gsiftp://roentgen-1.cs.umu.se:2811//home/michael/0a3fa010-a29d-11dd-aafe-8e00a8c2b63a' ->
  'download.txt'
ExtendedGridFTPHandler [main,reliableTransfer:237] 1. attempt to perform reliable SERVER_TO_CLIENT transfer.
  Transfer [main,setState:89] State transition: PENDING -> ACTIVE
  Transfer [main,setState:89] State transition: ACTIVE -> DONE
ExtendedGridFTPHandler [main,reliableTransfer:275] Final state of transfer: DONE
StorageBrokerImpl [main,downloadToFile:204] File successfully downloaded from
  'gsiftp://roentgen-1.cs.umu.se:2811//home/michael/0a3fa010-a29d-11dd-aafe-8e00a8c2b63a'
StorageBrokerImpl [main,downloadToFile:218] 3. Verify checksum
StorageBrokerImpl [main,checkDataConsistency:277] 1. Look-up in Catalog
StorageBrokerImpl [main,checkDataConsistency:288] 2. Verify checksum (SHA=1-hash,
  '10b1f8e6a5526e07c3e87b5f0e1acc3e3fa403')
StorageBrokerImpl [main,checkDataConsistency:291] Verified data consistency:
  '10b1f8e6a5526e07c3e87b5f0e1acc3e3fa403'
StorageBrokerImpl [main,downloadToFile:220] Verified data consistency.
```

The file '730e7c70-a29c-11dd-bcef-cf721e03f4f4' is downloaded from the Grid to 'download.txt'.

```
michael@roentgen-4:/tmp$
```
D.5 Remove a File

We delete the file from the Grid using the `dps-rm` command.

```
michael@roentgen-4:/tmp$ dps-rm 730e7c70-a29c-11dd-bcef-cf721e03f4f4
```

```
dps-rm DPS client v.1.0.1
CatalogService : https://roentgen-4.cs.umu.se:8443/wsrf/services/CatalogService
StorageBrokerService: https://roentgen-4.cs.umu.se:8443/wsrf/services/StorageBrokerService
Logical File Name : 730e7c70-a29c-11dd-bcef-cf721e03f4f4
```

```
StorageBrokerImpl [main,deleteFromGrid:232] 1. Look-up in Catalog
StorageBrokerImpl [main,getCatalogServiceInstance:99] Connect to CatalogService
StorageBrokerImpl [main,delegateCredentials:140] DONE
StorageBrokerImpl [main,deleteFromGrid:247] 2. Remove file
RFTServiceConnector [main,deleteRequest:130] Generating complex delete request.
RFTServiceConnector [main,rftRequest:200] Subscribed for overall status
RFTServiceConnector [main,rftRequest:206] Termination time to set: 60 minutes
RFTServiceConnector [main,rftRequest:220] Starting RFT
RFTServiceConnector [main,rftRequest:244] RFT finished.
RFTServiceConnector [main,rftRequest:246] All Transfers are completed
```

```
GridFTPHandler [main,exists:195]
Exists: 'gsiftp://roentgen-1.cs.umu.se:2811//home/michael/0a3fa010-a29d-11dd-aafe-8e00a8c2b63a''
StorageBrokerImpl [main,deleteFromGrid:255] File successfully deleted from
'gsiftp://roentgen-1.cs.umu.se:2811//home/michael/0a3fa010-a29d-11dd-aafe-8e00a8c2b63a'
```

```
GridFTPHandler [main,exists:195]
Exists: 'gsiftp://roentgen-2.cs.umu.se:2811//home/michael/759d1690-a29c-11dd-aafe-8e00a8c2b63a'
StorageBrokerImpl [main,deleteFromGrid:255] File successfully deleted from
'gsiftp://roentgen-2.cs.umu.se:2811//home/michael/759d1690-a29c-11dd-aafe-8e00a8c2b63a'
```

```
StorageBrokerImpl [main,deleteFromGrid:255] File successfully deleted from
'gsiftp://roentgen-3.cs.umu.se:2811//home/michael/0a56d190-a29d-11dd-aafe-8e00a8c2b63a'
```

```
StorageBrokerImpl [main,deleteFromGrid:255] File successfully deleted from
'gsiftp://roentgen-4.cs.umu.se:2811//home/michael/0a69bd50-a29d-11dd-aafe-8e00a8c2b63a'
```

```
StorageBrokerImpl [main,deleteFromGrid:271] 3. Update Catalog
```

The file '730e7c70-a29c-11dd-bcef-cf721e03f4f4' is removed from the Grid.

```
michael@roentgen-4:/tmp$
```

We try to download the file from the Grid once again, which is not possible as we deleted the file.

```
michael@roentgen-4:/tmp$ dps-cp-to-file 730e7c70-a29c-11dd-bcef-cf721e03f4f4 download.txt
```

```
dps-cp-to-file DPS client v.1.0.2
CatalogService : https://roentgen-4.cs.umu.se:8443/wsrf/services/CatalogService
StorageBrokerService: https://roentgen-4.cs.umu.se:8443/wsrf/services/StorageBrokerService
Logical File Name : 730e7c70-a29c-11dd-bcef-cf721e03f4f4
```

```
StorageBrokerImpl [main,downloadToFile:188] 1. Look-up in Catalog
StorageBrokerImpl [main,delegateCredentials:97] Connect to CatalogService
StorageBrokerImpl [main,delegateCredentials:130] Delegating user credentials ..
StorageBrokerImpl [main,downloadToFile:188] DONE
```

```
Something went wrong while downloading:
LFN '730e7c70-a29c-11dd-bcef-cf721e03f4f4' not found.
michael@roentgen-4:/tmp$
```

We try to download the file from the Grid once again, which is not possible as we deleted the file.

```
michael@roentgen-4:/tmp$ dps-cp-to-file 730e7c70-a29c-11dd-bcef-cf721e03f4f4 download.txt
```