UNIVERSITY OF SOUTHAMPTON

Distributed Data Management for Large Scale Applications

by

Miguel Branco

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosophy

in the
Faculty of Engineering, Science and Mathematics
School of Electronics and Computer Science

November 2009

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY of ENGINEERING, SCIENCE and MATHEMATICS
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Miguel Branco

Improvements in data storage and network technologies, the emergence of new high-resolution scientific instruments, the widespread use of the Internet and the World Wide Web and even globalisation have contributed to the emergence of new large scale data-intensive applications.

These applications require new systems that allow users to store, share and process data across computing centres around the world. Worldwide distributed data management is particularly important when there is a lot of data, more than can fit in a single computer or even in a single data centre. Designing systems to cope with the demanding requirements of these applications is the focus of the present work.

This thesis presents four contributions. First, it introduces a set of design principles that can be used to create distributed data management systems for data-intensive applications. Second, it describes an architecture and implementation that follows the proposed design principles, and which results in a scalable, fault tolerant and secure system. Third, it presents the system evaluation, which occurred under real operational conditions using close to one hundred computing sites and with more than 14 petabytes of data. Fourth, it proposes novel algorithms to model the behaviour of file transfers on a wide-area network.

This work also presents a detailed description of the problem of managing distributed data, ranging from the collection of requirements to the identification of the uncertainty that underlies a large distributed environment. This includes a critique of existing work and the identification of practical limits to the development of transfer algorithms on a shared distributed environment.

The motivation for this work has been the ATLAS Experiment for the Large Hadron Collider (LHC) at CERN, where the author was responsible for the development of the data management middleware.

Contents

List of Figures

List of Tables

Listings

Declaration of Authorship

I, Miguel Sérgio de Oliveira Branco, declare that the thesis entitled *Distributed Data Management for Large Scale Applications* and the work presented in it are my own. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as references [28], [30], [29].

Signed:

Date:

Acknowledgements

Every thesis is unique. But allow me to claim this uniqueness title as well. Throughout these past years, I was given a privileged opportunity: to work in a first-class research laboratory and pursue a PhD in a first-class university in another country. Yes, the constant plane trips were really fun early on (and really boring later on), but only now I begin to truly appreciate the opportunity I've been given. For this I have first to thank two people: Luc Moreau and Markus Elsing. Luc, although this thesis didn't really work out as we both expected, I have to thank you for first exposing me to what research is all about. Markus, the manner by which you get people excited and moving continuous to surprise me. I can't thank you enough for giving me the opportunity to work on such a relevant problem, and for the outstanding support you showed throughout the years!

Then there's the best supervisors I could have wished for: Ed Zaluska and David de Roure. From Ed I learnt many many things but let me single one out: the attention to detail! This is proving to be a very powerful tool and it was certainly a required one to reach this phase. The care you put on your students is something that inspired me, and I hope someday to pass it on! Dave, first and above all, thank you for the opportunity. You gave me "research shelter" when I was adrift! Be it in the confidence you instilled in me, or in the occasional informal conversations, I see all attributes of a great visionary, and I feel privileged to have worked with you! (And thank you also to your wife, Gillian, for editing this thesis!)

Even if this thesis is my own, much of its implementation has been a team effort. Many thanks to the DDM guys throughout these years: David Cameron with whom I started the project and found out what data management was all about, Pedro Salgado for his great (and substantial!) work and discipline, and to Mario Lassnig and Vincent Garonne with whom I spent my last years in ATLAS, sharing the occasional successes and the many frustrations :) Mario, Vincent, you guys kept me inspired, and I couldn't have wanted better office mates: but perhaps a few more end-of-afternoon beers would have been a good idea...

But it was not only the DDM team: it was also Torre Wenaus, Luc Goossens, Alexei Klimentov, Dario Barberis, Gilbert Poulard, Kaushik De, Armin Nairz, Massimo Lamanna, Simone Campana, Stephane Jezequel, Graeme Stewart, Kors Bos, Ricardo Rocha and Gavin McCance (and many others!). Torre, you've been extremely influential in this work, in my career and even to me personally, and for that I thank you. Luc, also one big thank you: whatever the coincidences that led me to CERN, it's you I have to thank first for all these opportunities and for keeping me grounded. Alexei, thank you for keeping up with me and my work throughout these amazing years: I learnt a tremendous deal from you. Dario, thank you for your constant support and investment in my development. Gilbert, you were the first person I met at CERN and indeed it was a great way to start: thank you also for this opportunity. Kaushik: I guess it's only you left from the early team now; thank you for the confidence and for the good humour! Armin, you've been a really great office mate and a terrific friend. Hope we get to meet often because I enjoy that subtle but great humour. Massimo: thanks for putting up with me :) Looking back, I have to remark the wisdom in your character: that ability to discuss and disagree yet listen, to influence yet compromise. I really learnt a lot from you. Simone, Stephane, Graeme, people, thanks! Thank you for being the first to help out after my occasional (coff coff) operational screw ups and for that outstanding ability to maintain good humour under stress. Kors, one sentence: I want to be like you when I grow up (plane trips included :) Outstanding manager and what an inspiration you have been! Rocha: who would have though that car trip to Switzerland would get this far? And Gavin, many thanks particularly in giving me access to much of the required data for this thesis; it proved very important.

Life really didn't stop at work though, and there's many friends to thank for that. Cesar for his really bad jokes that made me run back to the office, Ricardo Silva for the widest smile on the earth, Miguel Santos for his only slightly better jokes :), Parracho, Martins and Moreira for great friendship, Frodo whose geekness was welcomed during the lunch, to Bastos and Vanita for keeping me social and even somewhat trendy, to Miguel Anjo for keeping me in shape, to Paulo Campos whose caipirinhas got me out of shape, and to Huguinho and Melo for... humm... I'm sure there was something... Oh yes, and to Elias for his insanity and dating tips: they proved valuable :) Then there's Pedro and Teresa; that would require an acknowledgement page of its own so let me keep it short: *Obrigado!* And thank you to my big brother, Pedro Branco, which has been my biggest inspiration to obtain a doctoral degree in the first place (and not to mention, go into computer science). But if I have to say only one word, let it be the one that really matters: *Anki!* Thank you. This wouldn't have happened without you and without your example; many things wouldn't have, but I'm really happy they did. Thank you for making these great times... but the best times are just starting!

Para os meus pais, por tudo!

Chapter 1

Introduction

Let us start with two examples.

A distributed team of scientists collaborate in a scientific experiment. The scientists work for different research centres, located in several countries. The research centres have joined to create a large multi-national scientific experiment, which shares each centre's computational resources across national, geographical and political borders. The question now is to develop a system that allows these collaborations to operate, and in particular for scientists to share and collectively work on their scientific data undisturbed by political or technical constraints.

A large commercial company provides services to customers around the world. It has built several data centres throughout the past years, and as the technology evolved, these data centres have become more heterogeneous, relying on different and sometimes newer technologies. In addition, the recent acquisition of a competitor brought in additional data centres, yet again with different internal architectures. The company's core business requires the ability to store and process vast amounts of data collected regionally, as well as the sharing of data between company analysts worldwide. The challenge now becomes how to develop a data management system that allows for massive data processing in a worldwide distributed and heterogeneous computing environment.

The emergence of large scale data-intensive applications illustrated in the previous examples is not a new trend but has been greatly amplified in recent years. The improvements in data storage technologies, the development of new high-resolution scientific instruments, the widespread use of the Internet and the World Wide Web and increasing globalisation have all been contributing factors.

For these applications data is the core component. In fact, a recent publication discusses "The Unreasonable Effectiveness of Data" [85], where access to very large amounts of noisy, unlabelled and human generated data can be used to build high-quality models in a very effective manner and embrace the complexity of some domains.

Others go further, signalling the "The End of Theory" and how the data deluge can make the scientific method obsolete [8]. Even without such a radical modification to the philosophy of science, clearly increased data processing capabilities have been remarkably useful. These increased capabilities have allowed new businesses to merge, such as Google, and new experiments with complex computational requirements to be realised, such as the Large Hadron Collider project at CERN.

But for data to be useful, systems must exist that allow users to store, share and process the data. In particular when there is a lot of data, more than can fit on a single computer or even in a single data centre. Designing such systems is the focus of this work.

Unfortunately, even if the modern experience of using a computer for analysing data can be less than pleasant, then to rely on a large set of distributed computers to store and process data can be far worse:

> "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."
> *(Leslie Lamport, email exchange in 1987)*

Although the well-known quotation above dates from 1987, few would disagree that it still remains applicable today. This is one of the problems addressed in this work. In addition to masking failures and addressing the complexity of distributed systems, this thesis also solves new issues that originate from widely distributed environments, such as how to organise very diverse storage resources into a single and unified user interface.

Extensive work has already been undertaken into this problem by a number of different researchers in the context of distributed file systems, peer-to-peer, grid or cloud computing. These contributions are analysed in Chapter 2. The work in this thesis builds from those foundations and defines a new application-layer middleware for managing distributed data. Before introducing these previous contributions, the next section describes the overall problem of managing distributed data in more detail.

1.1 The Problem of Managing Distributed Data

The shift from using a single computer to using sets of distributed computing resources resulted in the creation of a new domain of computer science, which is *distributed computing*. Distributed computing has enabled new methods for solving increasingly complex problems by dividing problems into smaller tasks that run on a number of remote hardware or software systems. Many important results have been established, from the creation of the ARPANET in the late 1960s, to the first widely used distributed file system [151], to theoretical results that establish the limits of asynchronous consensus algorithms under unannounced crashes [71].

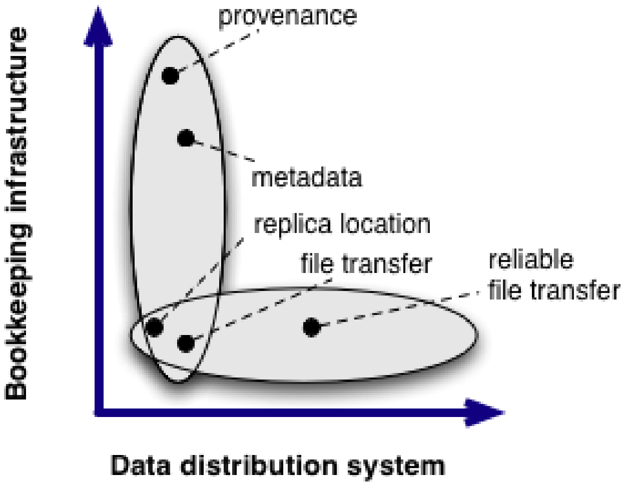


FIGURE 1.1: Representation of the domain coverage of existing distributed data management contributions.

More recently a new approach to distributed computing has emerged, which is characterised by the existence of distributed resources around the world. Dedicated or shared data centres are used by multiple organisations, enabling access to a much larger set of resources. This new paradigm provides new opportunities but also raises a set of challenges, some of which are shared with classical distributed computing, while others are unique to this new environment.

A new opportunity concerns failure handling, which has always been an integral part of distributed computing. Worldwide distributed computing allows for more robust solutions. Companies can now protect their data from catastrophic failures caused by accidents (fires), natural disasters (earthquakes, floods) or even terrorist attacks by distributing their data around the world. Worldwide distributed computing also enables greater sharing of resources and the formation of new collaboration models, where several entities join existing resources into a single set, increasing the computational power available to all parties.

Unfortunately worldwide computing also creates new problems and more complex versions of existing problems. For instance, latency has always been a limiting factor when distributing a task between several computers. But when there is a dependency on resources around the world, the physical limits imposed by the speed of light become a much greater restriction. Latency is no longer an annoyance but can become a major bottleneck and design restriction. For instance, the Google developers have conditioned their Google App Engine data store transaction model more severely than they desired, due to the latency in the communication across data centres[1].

The goal in this thesis is to address an (important) subset of these problems. They are

[1] Refer to *http://code.google.com/events/io/sessions/TransactionsAcrossDatacenters.html.*

related to the management of distributed data. I start by introducing my definition of *distributed data management*, as the computer science domain describing methodologies applied to applications running on distributed systems where application data is the primary resource in the system. A distributed data management system can be represented by two main axes, as shown in the Figure 1.1. These are the *Bookkeeping infrastructure* and the *Data distribution system.*

The vertical axis of the figure corresponds to the bookkeeping infrastructure, which is responsible for handling all aspects related to the bookkeeping of data. These range from tracking locations of individual data files in the geographically-distributed data storages[2] and knowing the properties of each file (e.g. file size or access control lists) to higher-level user or domain-specific metadata properties. Additional examples are shown in the figure. The bookkeeping infrastructure is also responsible for establishing the organisational infrastructure of the data, which defines the relationship between data files, including for instance the provenance relationships in the data.

The other axis in Figure 1.1 is the data distribution, which uses the information from the data bookkeeping layer and handles, potentially in an automated way, the distribution of data around data centres. This involves interactions with data transfer services and storages, and movement of data between (typically wide-area) networks. The data distribution axis also includes the enforcement of quality of service in the data transfers and issues such as security, consistency of data, monitoring and recovery of failed transfers. The figure attempts to illustrate the coverage of existing work on both bookkeeping and data distribution, which is marked in the "grey" areas of the picture and where "richer" services for end-users are represented in the topmost and rightmost portions of the graph.

Looking at Figure 1.1, there is a portion of the domain that is not covered. This is the gap in the top-right portion of the graph, which corresponds to contributions that would tightly integrate the two aspects of distributed data management. This part of the domain is nowadays typically covered by end-user applications that create application-specific distributed data management systems. This in a sense is the distributed data management problem, which is to create general solutions to cover this and the remaining domains, as opposed to the repeated development of custom solutions.

I now introduce Figure 1.2 that illustrates the coverage of the distributed data management system introduced in this thesis. From the perspective of an application or end-user, it is precisely the convergence between the two axes that produces the most functional system. This is represented in the figure as `Area X`, which is where the system functionality is considered richer.

[2] In this thesis, the terms *storages* and *storage systems* are used interchangeably. A storage system can be loosely defined as a container of user data, be it in a file system, relational database or other forms.

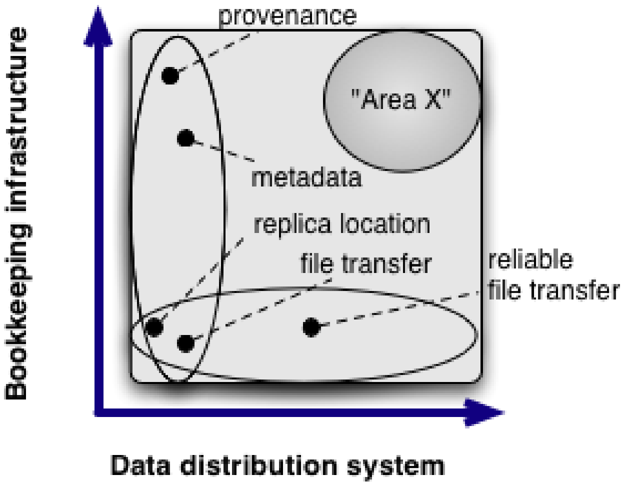


FIGURE 1.2: Representation of the intended domain coverage for a new distributed data management system.

A key objective is that the data distribution must attempt to make optimal use of the bookkeeping properties of the data. This allows the system to provide a given quality of service and address issues such as consistency of data in the distributed system by exploiting the bookkeeping structures for data distribution.

As an example, when an end-user needs to locate a piece of data, the query often starts with a metadata query. This (high-level) metadata query - e.g. "retrieve all data that has properties X, Y and Z, which I require for my analysis" - results in a set of data items whose properties, such as the physical location, are tracked by the bookkeeping system. The user does not know directly what are the data items or where they are physically located. The distributed data management system, by linking both the bookkeeping and data distribution functionalities, can serve the user request, translating high-level metadata queries to physical files for transfer by the data distribution system.

Similarly, any consistency issues encountered during regular activities - e.g. a transfer failure is determined by the data distribution system to be caused by a lost file - must be propagated up from low-level file transfer layers to the higher-level monitoring services also part of the data distribution system. In addition, these occurrences must also be communicated to the bookkeeping system so that users are informed of potential data inconsistency issues even in response to high-level metadata queries.

There is a raft of technologies that partially provide the required functionality. However, existing technologies fail in one fundamental aspect: they are not integrated together to form a complete system. Instead, these technologies provide a set of rather loosely-coupled components, which are provided to the applications as building-blocks for building application-specific systems.

1.2 Theory and Practice

Several research areas have contributed to the distributed data management problem. The data management theory provides many of the underlying principles, such as the definition of data and consistency models. Within distributed computing, there has been significant work on distributed file systems, peer-to-peer systems, grid and cloud computing. The relevant contributions are discussed in Chapter 2.

In addition, specialised areas within theoretical computer science have developed algorithms for routing (e.g. packet routing), data placement (i.e. how best to place data given expected access patterns) and data migration (i.e. how best to change from one placement layout to another).

Nonetheless, several open questions remain.

- What are the exact requirements that define the distributed data management problem?
- Is it possible to define a set of high level design principles that can be used to create a distributed data management system that is more general-purpose and feature-rich than existing contributions?
- Can the resulting system be demonstrated to work under real operational conditions, managing petabytes of data for a real application?

In addition, there are some more specific questions that are of interest:

- Is it possible to design a distributed data management system that can be easily integrated with an existing application without enforcing significant changes to that application?
- In addition, is it reasonable to maintain or augment the file *modus operandi*, in which users create and share data files in the wide-area network without having to move the data onto new systems (such as relational databases) that employ different data models?
- What are the practical limits to the development of algorithms for data placement and data migration? In particular, what is the impact of real operational conditions on the distributed data management system?

The next section follows from these open questions by discussing the contributions presented in this thesis.

1.3 Thesis Contributions

The contributions in this thesis are:

- The introduction of a set of design principles that can be applied to the creation of a non-intrusive and scalable distributed data management system.
- The description of an architecture and implementation that follows the proposed design principles, and which results in a scalable, fault tolerant and secure system.
- The evaluation of the proposed system in real operational conditions, which include using close to one hundred computing sites and the management of several petabytes of data.
- The creation of novel algorithms to model the behaviour of file transfers on a wide-area network. The resulting models have the advantage of being simpler to apply compared to existing simulation tools.

Throughout this thesis I also present a more accurate description of the problem of managing distributed data in real operational systems, ranging from the collection of requirements to the identification of the uncertainty that underlies a large and distributed environment. Therefore, a very important implication of this work is an analysis of systems that rely on predictive frameworks and simplistic infrastructure models. This thesis demonstrates that the real world is significantly complex and that important theoretical challenges remain. As such, the discussion on future work directions is also an important contribution as it presents tentative work directions to address these issues in the future.

1.4 Presentation Overview

Chapter 2 ("Literature Review") reviews existing contributions that are closely related to the distributed data management problem. This includes systems developed within the domains of distributed file systems, peer-to-peer systems, grid and cloud computing. Several systems developed for specific scientific applications are also discussed.

Chapter 3 ("System Motivation") discusses a specific distributed data management problem, which provides a focus for the research. In this chapter, a large scale application - the ATLAS Experiment for the Large Hadron Collider at CERN - is described in detail. The ATLAS Experiment has served as the primary motivation for this work, but its requirements and data flow include common topics with other data-intensive applications, which are also discussed in the chapter.

Chapter 4 (“System Requirements and Architecture”) introduces the set of generic design principles for the distributed data management system. The chapter discusses the methodology taken as well as the system requirements, which are defined for users, managers and data centre administrators. It also includes a high-level overview of an architecture that follows from these generic design principles.

Chapter 5 (“System Design”) describes the implementation of the system in greater detail. The presentation is organised by each of the components in the system. For each component, the relevant behaviour and properties are described, in addition to the algorithms, schema and interfaces. This is done in sufficient detail to allow the work to be replicated.

Chapter 6 (“System Evaluation and Simulation”) starts by presenting the usage of the system for a real world application. This is followed by substantial evidence of its scalability and performance. The second part of Chapter 6 introduces new modeling techniques. These techniques result from studying some of the observed behaviour as well as the need to develop a simulation framework for the system.

Chapter 7 (“Conclusion and Future Work”) is divided into three parts. In the first part, the contributions are revisited in more detail. The second part outlines future directions for this research, and the third part presents the traditional concluding remarks.

1.5 Publications

The following peer-reviewed publications include some of the content described in this thesis:

M. Branco, E. Zaluska, D. De Roure, P. Salgado, V. Garonne, M. Lassnig, R. Rocha. Managing Very-Large Distributed Datasets. In *Proceedings of the OTM Conferences*, volume 5331/2008 of *Lecture Notes in Computer Science*, pages 775–792. Springer, 2008. ([30])

This conference publication includes the architecture described in Chapter 4 and the implementation details described in Chapter 5.

M. Branco, E. Zaluska, D. De Roure, M. Lassnig, V. Garonne. Managing very large distributed datasets on a Data Grid. Accepted for publication in *Concurrency and Computation: Practice & Experience* (in press). ([29])

This journal publication includes a summary of Chapter 4 and Chapter 5, and describes the modelling and simulation techniques presented in Chapter 6.

M. Branco, L. Moreau. Enabling Provenance on Large Scale e-Science Applications. In *Provenance and Annotation of Data*, volume 4145/2006 of *Lecture Notes in Computer*

Science, pages 55–63. Springer, 2006. ([28])

This conference publication discusses how to enable provenance gathering in a large scale experiment, through the integration of the distributed data management system with external metadata systems and by using provenance recording protocols. These issues are discussed in Chapter 5 and Chapter 7.

Chapter 2

Literature Review

This chapter introduces previous research contributions. In the first section I start by introducing relevant contributions from the domain of distributed file systems. This is followed in the second section by peer-to-peer contributions and grid computing in the third section. Grid computing in particular presents many important similarities in the working environment and serves as the foundation for the system introduced in later chapters. I also describe recent work on cloud computing in the fourth section, which is interesting given its novel approach to data management issues. This is followed by implementation examples from several scientific applications. The final section summarises the important points from the literature and highlights the lessons learnt and applied to my work.

In each section, with the exception of cloud computing and scientific applications, I include a dedicated discussion on replica placement. There are two main motivations for this choice. First, there is a large set of important contributions that explicitly address replica placement issues in each of these domains. Second, understanding replica placement strategies often provides valuable insight into how the systems work internally.

In addition, there are several criteria used in the discussion of each contribution. These are: scalability, support for heterogeneous environments, support for quality of service guarantees, availability, reliability and fault tolerance.

Scalability is discussed both in terms of load (i.e. the ability to store and manage petabytes of data) and geography (i.e. the support for world-wide distributed storage). *Heterogeneous environments* are discussed because, as presented in Chapter 1, this work requires the ability to merge many different distributed storage systems into a single unified layer. Therefore, it is important to describe to what extent (and how) existing systems support this capability.

The third criteria concerns the *support for quality of service guarantees.* This support is necessary given the goal to provide more advanced data management functionality, as represented by the desired domain coverage in Figure 1.2 of Chapter 1. Quality of service guarantees are loosely defined as the properties that allow users control over the behaviour of the system and include, for instance, the ability to establish the relative importance of competing requests.

The remaining criteria are important properties for any distributed system. *Availability* (i.e. the degree to which the system is operable when its functions are requested), *reliability* (i.e. the ability of the system to perform its functions under stated circumstances) and *fault tolerance* (i.e. the properties that enable the system to continue operating properly in the event of failures) are therefore discussed in the following analysis.

In particular, the discussion on fault tolerance encompasses both byzantine and non-byzantine fault tolerance. Byzantine fault tolerance is the ability to protect against a byzantine failure, which occurs when a component not only behaves erroneously but fails to behave consistently. This type of "failure" is characteristic of environments where malicious attacks can occur. In this thesis, the focus is not on malicious environments: the assumption is that reasonable protection mechanisms can be put in place to protect the communication between components, but that deployed components are not altered in any malicious manner.

One area that is not included in the following review is the large set of theoretical contributions on network routing and data replication. Instead, the focus has been to describe complete middleware systems. When theoretical contributions have been integrated into existing systems these are discussed. Some additional theoretical contributions are included as part of future work directions in Chapter 7, where novel research directions will be proposed.

2.1 Distributed File Systems

In this section I introduce relevant work in the area of distributed file systems. I start by the most common variant, which are distributed file systems for clusters. This is followed by a few examples of file systems for wide-area networks. I then present additional details on the replica placement algorithms used in these systems and present conclusions.

2.1.1 Cluster File Systems

NFS [151] is one of the early distributed file systems and continues to be widely used. It is based on a stateless (up to version 4) client/server protocol implemented using remote procedure calls and supports POSIX-like semantics. Experience has established that this is a reasonable goal on fast networks without many clients and simpler usage patterns [27]. However, with large numbers of users or with bandwidth constraints, the

POSIX-like semantics hinder the performance and scalability, resulting in NFS being an unattractive choice to manage datasets at the petabyte scale.

AFS [89] was the first distributed file system to introduce caching of file and directory information on the client machine. This property increases the scalability of AFS but introduces additional complexity when handling updates. UNIX “last file write wins” semantics are especially hard to implement in a scalable manner. To address this, AFS introduces “last file close wins” semantics, where the last file close operation becomes immediately visible to other clients. This limits the universal applicability of AFS to general work patterns but increases the scalability for the common cases of multiple reads with infrequent writes. Nonetheless, in the literature I have not found any AFS-based system handling petabytes of data over a wide-area network. This is likely due to fundamental AFS design decisions, such as the requirement to support POSIX-like semantics. There are two important lessons to be learnt from the AFS design: the scalability benefits from client-side caching and the need to adapt the file-locking semantics for specific usage patterns.

Coda [152] is a file-system originally based on AFS that introduces support for disconnected operations when the network connection is lost. Coda provides mechanisms to resolve conflicts. In addition, while AFS uses pessimistic replication, allowing only one read/write server to receive updates and other servers to act as read-only replicas, Coda allows all servers to receive updates, increasing scalability and providing greater protection against server failures. Nonetheless, at the petabyte-scale, Coda suffers from exactly the same issues as AFS.

Another modern cluster file-system is the block-based IBM General Parallel File System (GPFS) [153]. GPFS provides high-performance I/O due to its ability to stripe blocks of data from individual files over multiple disks, as well as reading and writing these blocks in parallel.

The Lustre [155] distributed file system was originally developed by Cluster File Systems and later acquired by Sun Microsystems. Lustre is inspired by the (very clean) architecture devised for the Digital VAXClusters, which were built on top of a local file system by requiring data access to interact closely with a distributed lock manager. The core components of Lustre are the distributed lock manager, the metadata servers and object storage targets. By exploring a well-designed distributed locking manager and caches, Lustre scales to the data handling requirements discussed previously: tens of thousands of nodes and petabytes of storage. Lustre is widely used in both commercial and scientific environments and is the backend to several supercomputers.

An important lesson from Lustre is on how scalability is achieved by moving from a block-based approach to an object-based approach, which changes the fundamental mechanisms used to access data. Lustre, contrary to traditional block-based devices, assumes that storage devices are intelligent devices and makes use of more advanced protocols to access data. Lustre clients do not talk directly to the block-based device but rather to a component called Object Storage Target (OSTs). This approach eliminates many of the bottlenecks of traditional block-based I/O in the communication between clients and block-based storage devices.

Although object-based approaches induce scalability gains, there is still a problem in scaling up the metadata access: in principle, adding OSTs allows the storage to scale almost linearly but the required metadata (e.g. directory information) does not. The Ceph file system [171] aims to address this problem by replacing traditional allocation and inode tables with a pseudo-random data distribution function, designed for dynamic clusters. It also uses a new reliable object storage called RADOS [173]. Overall, Ceph achieves better scalability by reducing the bottlenecks associated with maintaining metadata on a distributed file system.

Google has designed and implemented the Google File System (GFS) [80], which provides a scalable system for distributed data-intensive applications. It is designed for applications handling very large files with many reads and few writes. GFS drops some of the assumptions of the earlier systems, such as POSIX-like semantics. It consists of a master node (the metadata server) and multiple chunkservers. The master node maps a user file to multiple chunks (a chunk is a block of 64 Mbytes), which are placed in various chunkservers. The file system supports parallel read, write and update operations and has built-in fault tolerance features.

This simple design allows GFS to scale to large clusters while running on inexpensive commodity hardware. Hadoop HDFS [1], a top-level Apache project, is a system inspired by the design of GFS but Open Source and therefore considerably better documented. HDFS implements some additional functionality such as the ability to expose its data through the WebDAV protocol [66].

A core lesson from these systems is that scalability is achieved by taking advantage of environment constraints. For instance, GFS eliminates the complex distributed locking models of the earlier systems by allowing append operations only and adopting simple mechanisms for fault tolerance.

Farsite [3] is a distributed file system with the goal of ensuring secrecy of file contents by using cryptographic techniques. It maintains the integrity of file and directory data, employs local caching, lazy update propagation, and varying content and duration of leases. Farsite is not meant for high-performance I/O or wide-area environments but for local-area networks with reduced write sharing scenarios. Its interesting properties lie in the mechanisms employed to achieve fault tolerance and data protection.

[1] Refer to *http://hadoop.apache.org/core/docs/current/hdfs_design.html.*

2.1.2 Wide-area File Systems

A major goal in this research is efficient operation over wide-area environments. In this section I describe existing work that customises or builds file systems for the wide-area network.

LegionFS [175] is an object-based system for the wide-area network, with the goal of supporting heterogeneous environments. It is an object-based system where objects implement a set of characteristics and exchange messages. It does not handle replication or load balancing but instead provides interfaces and protocol stacks that can be extended by application developers. The scalability it achieves is limited because of the object-based approach and extensive use of messaging.

Lustre was initially designed as a cluster file system for a closed network but has since been expanding to accommodate deployments across clusters and data centres. [128] briefly describes future plans for a "Lustre Router Control Panel" to allow adjusting the quality of service within a cluster and wide-area network. [158] demonstrates a scenario where Lustre runs over several wide-area computing centres. This is done with dedicated 10 Gbit links and some tuning of the settings in Lustre.

GPFS has been demonstrated to work over a wide-area network in [9]. Nonetheless, this was done with strict deployment constraints and considerable tuning much like the previous example with Lustre.

WheelFS [162] is a wide-area distributed storage system that implements a POSIX interface. It allows applications to adjust the trade-off between prompt visibility of updates from other data centres and the ability for centres to operate independently despite failures and long delays, hence relaxing the stricter POSIX semantics. It relies on "semantic cues" that provide applications with control over consistency, failure handling, and file and replica placement. WheelFS has been implemented as a user-level file system but its evaluation focuses on smaller files (O(10) MBytes) and smaller data rates.

2.1.3 Replica Placement

Most of the systems previously presented, particularly earlier work, employ rather simple replica placement strategies. Some do not even employ any technique and leave the issue of availability to underlying RAID systems. Other systems employ only minimal replication: e.g. GPFS optionally creates two replicas of a file if RAID is not available.

The Google File System has a more complex algorithm. The replica placement in GFS tries to place replicas in uncorrelated devices, which are typically storage devices in different racks. This strategy is clearly optimised for a data centre environment. Its architecture presents a decisive advantage for replica placement: the master-based architecture allows GFS to achieve close to ideal resource balancing, since placement decisions can use global knowledge. Nonetheless, the master-based architecture can result in a scalability problem and in a single point of failure. Additionally, the master needs to detect and react quickly to failures in all its storage nodes.

The Ceph file system introduces an interesting replica placement strategy. This is called CRUSH [172], a scalable pseudo-random data distribution function designed for distributed object-based storage systems that does not rely on a central directory. It allows dynamic addition and removal of storage devices. It also accommodates a variety of data replication and reliability mechanisms and distributes data in terms of user defined policies that enforce separation of replicas across failure domains.

CRUSH places objects in storage devices according to a per-device weight value, approximating a uniform probability distribution. This distribution is controlled by a hierarchical cluster map. The data distribution policy is defined in terms of placement rules that are applied onto the cluster map. The difficulty in CRUSH is handling changes to the hierarchical cluster map as these may lead to significant data reshuffling. In addition, CRUSH's cluster map is designed for a cluster environment and provides no specific hooks for wide-area environments.

Kinesis [122] is another recent replica placement algorithm used for distributed file systems. Unlike CRUSH, Kinesis has not yet been applied to an existing distributed file system. Kinesis divides servers into a few failure isolated segments, following the principles of GFS or CRUSH. Similarly to CRUSH, it also makes use of pseudo-random functions to spread replicas in the system. Unlike CRUSH, Kinesis allows some freedom of choice when placing replicas.

In Kinesis the storage servers are partitioned into disjoint segments of approximately equal size. Servers with correlated failures (e.g. those on the same rack) are assigned to the same segments. An independent hash function is associated with each segment and maps identifiers for data items to servers in the segment. The hash function dynamically adjusts to accommodate for changes in the set of servers due to failures or addition of servers.

Kinesis makes use of the multi-choice paradigm [13] when choosing the segments to place a replica. Within each segment Kinesis uses linear hashing [117]. These decisions ensure that the algorithm maintains a balanced resource utilisation even in moderately dynamic environments. Nonetheless, like CRUSH, it has no specific functionality for the wide-area network (e.g. knowledge of network latency for replicating data), as it assumes a data centre environment and makes simplistic assumptions to compensate for possible heterogeneity of the underlying storage resources.

2.1.4 Discussion

In the literature, there is no file system with native support for heterogeneous systems or the wide-area environment. Nonetheless, most active projects have ongoing work intended to extend to a wide-area environment (e.g. GPFS, Lustre). Currently, such proposals require manual tuning between any two hops but these developments confirm that the goals discussed in this work are relevant.

There are several common architectural design decisions adopted in the systems discussed above. One is that metadata is handled by a separate service (e.g. GFS, Lustre or CRUSH). Even though a central metadata service (such as in GFS) is sufficient for most usages, this approach has potentially limited scalability. Another observation is that the more recent systems do not store user files as individual files on the storage (e.g. Lustre). GPFS, GFS and Hadoop also adopt a similar approach: files on the disk servers have no direct correlation with user files. Finally, most distributed file systems maintain at most POSIX-like semantics and systems such as GFS or Hadoop are not POSIX compliant at all. This is done to increase scalability. Apparently, users have no significant difficulties in adapting their systems to these new interfaces.

Replica placement algorithms provide an interesting insight into how distributed file systems actually work. Nonetheless, existing algorithms are designed for a data centre environment (e.g. CRUSH requires a cluster map and Kinesis requires the definition of failure isolated segments). As such, these systems make simplistic assumptions regarding network connectivity, do not deal with transfer latency and have simple mechanisms for detection of faulty nodes, often building on top of data centre specific tools.

2.2 Peer-to-Peer Data Management

Peer-to-peer (P2P) systems are based on decentralised architectures [165] where, from a high-level perspective, all processes in the system are equal and their interactions symmetric. Each process can act both as a client or server at the same time. In addition, P2P systems implement self-organisation and distributed control [148].

P2P systems are interesting for their scalability and associated fault tolerance properties. An example of a P2P system in widespread usage is BitTorrent [48], a file distribution system that in 2007 was responsible for approximately 70% of overall Internet traffic and accounted for over 66% of the P2P user population [91].

In the next section I describe how the processes in a P2P system are organised into overlay networks in order to understand how scalability is achieved. This is followed by a description of P2P systems used to store and share files among users. Replica placement strategies are discussed in the third section. Finally, I conclude with a discussion of the relative advantages and disadvantages of each system.

2.2.1 Overlay Networks

Participating processes (or nodes) in the P2P system are organised to form an overlay network, which is a virtual network with a link between any two nodes that communicate directly with each other. There are multiple types of overlay networks identified in the literature [120]. These overlay networks have been classified as centralised or decentralised, in which case they can be considered as structured or unstructured [121].

Early systems, such as Napster[2], were based on a constantly updated centralised directory. This central directory was used to find other nodes: in the case of Napster it was used to locate nodes that had the desired music or video files. Nonetheless, this approach does not scale well since it contains a single point of failure and has since been mostly abandoned.

Decentralised but structured systems have no central server but implement a significant amount of structure. Literature shows that most such systems make use of distributed hash tables (DHTs). This allows for building efficient and deterministic mapping of keys to nodes based on some distance metric. Examples of DHT-based systems are Chord [161], Freenet [47], Pastry [149], Tapestry [180] and the Content Addressable Network (CAN) [144].

Decentralised and unstructured systems rely on randomised algorithms to build the overlay network. Each node maintains a list of its neighbours built in a more or less random manner. The final network has certain properties but there is no deterministic mechanism to locate nodes (e.g. based on some key as in DHT-based systems).

The presence or absence of structure in the overlay network is very influential to the design of P2P applications. The most common P2P application is a file distribution service: this involves searching and transferring a file to a user node. In the presence of structured systems, searches can make use of structural information. In the case of unstructured systems, a query usually involves flooding the network [147], which may impair its scalability for large numbers of nodes.

Recent literature shows a prevalence of some form of structure in most systems, following work by [39]. Note that the classification between structured and unstructured is not entirely strict and [87] has introduced a P2P classification based instead on the existence and location of an index, following classical database research terminology.

In the next section I describe relevant work on P2P storage systems.

[2] Refer to *http://www.napster.com.*

2.2.2 Storage Systems

P2P systems became popular through file distribution applications where users share music and video files. The inherent properties of P2P systems allowed for significant scalability improvements. These were quickly exploited by the research community, which followed and improved on this trend by introducing distributed storage systems. Some early examples are CFS [52] and PAST [65], which both make use of (structured) overlay networks. In the case of CFS, Chord [161] is used for the overlay network. Other examples are OceanStore [102] (and its prototype Pond [146]) and Ivy [130]. These systems target a wide-area environment and have in common the use of distributed hash tables. In the case of OceanStore, the authors aimed at designing a wide-area network system capable of storing all the World's data.

One of the major issues with early P2P storage systems was the lack of any quality of service guarantees. This is addressed in [25] under the suggestive title "High availability, scalable storage, dynamic peer networks: pick two". One influential factor is called *churn*, which is the impact on the system of constant changes in the set of participating nodes due to joins, graceful leaves, and failures [82]. This is particularly problematic for DHT-based systems, which may require repairing hash tables.

An attempt to solve the lack of service guarantees is provided by TotalRecall [24]. TotalRecall is a P2P storage system that automatically manages availability in a dynamically changing environment. It adapts the degree of redundancy and frequency of repair to the distribution of failures. As such, it can guarantee user-specified levels of availability while attempting to minimise the overhead needed to provide these guarantees. Carbonite [46] claims to improve on this principle by devising a new algorithm that requires substantially less network overhead and still guaranteeing data durability.

A slightly different approach is taken by Kosha [34]. Kosha is an P2P enhancement to the NFS file system and is built on top of Pastry. It allows for distribution of directories instead of files across multiple nodes, reducing the NFS-related overhead.

More recently, some researchers have focused on creating P2P-based solutions for a more benign environment. In this environment, it is assumed that there is a better control (ownership) of the storage resources, hence less churn. In this scenario, it is possible to achieve a better load and storage balance. An example of such a system is BitVault [179].

2.2.3 Replica Placement

The P2P literature also shows extensive research on algorithms for replica placement. The goal of a P2P replica placement algorithm is to place data in the system in such a way that a query, propagated through broadcast or random walk, finds the data with a certain probability within a bounded search. This avoids the need for queries to span the entire network.

Work on replica placement was initially motivated by unstructured systems and the reference work is by [121] and [49]. The authors showed that the optimal search performance can be attained when the number of copies of a data item are proportional to the square root of its popularity. [115] shows how to implement such a strategy.

[129] introduced protocols that implement a random walk followed by a deterministic walk that ends in a local minimum for the current item. The result is exponentially increasing success probability as the number of replicas increases. [159] improves these contributions with provable results, adding the ability to deal with more dynamic churn situations and also addressing where to place replicas. It is worth noting that whilst most of this research is motivated by unstructured networks, it is also relevant for the structured case.

In the next section, I discuss the relative advantages of these contributions in more detail as well as their applicability to my problem.

2.2.4 Discussion

P2P systems are interesting because they address both the heterogeneity and the wide-area problems. These systems are layered on top of the existing file systems in each participant's computer, which are linked through overlay networks spanning the Internet, allowing users to share their data files. The fact that most P2P systems have evolved to have some sort of structure through overlay networks highlights the difficulties with working with a completely random environment.

Nonetheless, P2P systems come short of my stated requirement as they do not provide significant quality of service guarantees. P2P systems are designed for a very large number of participating nodes in a byzantine environment, while I aim for non-completely byzantine scenarios. This in part is the reason why P2P systems have limited ability to provide service guarantees. In addition, P2P systems for the most part support very limited functionality as I describe next.

PAST and CFS do not provide advanced data management capabilities. For instance, CFS assigns all objects the same number of replicas and disregards popularity skews, which may not be suitable for some applications where all data items are equally important and should be retrieved even if less popular. CFS has also has been documented not to have good performance [173] apparently due to its reliance on Chord. PAST also caches popular objects along the query path, which may cause storage utilisation imbalances: that is, some P2P nodes host data that they will never use, resulting in wasted disk space.

Ivy, which is built on top of DHash [53], does not provide for balanced storage utilisation either. It is reported to cause excessive bandwidth and storage waste by creating new replicas in response to transient failures [46]. This derives from the difficulties in distinguishing short-term transient failures from permanent failures where nodes permanently leave the network or lose data.

OceanStore is designed for a totally untrusted and constantly changing environment. Like PAST, CFS and Ivy, OceanStore only provides probabilistic guarantees. Unlike PAST and CFS, it provides for write sharing. As a result, it has the disadvantage of a complex architecture featuring a byzantine agreement protocol for conflict resolution and a complex protocol to implement the location service (the authors assumed the core system could be maintained by commercial providers). Its scalability has also not been evaluated at a very large scale.

Both TotalRecall and Carbonite are optimized for the wide-area network and attempt to reduce the number of replicas created by temporary failures. These systems essentially focus on data availability and not other classes of requirements, such as priorities in accessing data or more advanced data bookkeeping. Also, their scalability has not been extensively studied and the simulation results are for scenarios with a large number of nodes but a comparatively small number of stored files.

Kosha implements a simplistic replica placement strategy where replicas are maintained in neighbour nodes in a fairly random assignment. Neighbours in the node identifier space have no relationship in terms of physical proximity. As such, it is not clear whether Kosha can achieve adequate storage utilisation balance.

In addition, none of the previous systems fits very well in a more benign environment with less churn. One of the main reasons is that the replication of data is fully decentralised, being done sequentially from a root node. This fully decentralised approach leads to difficulties in attaining balanced storage utilisation ("global optimums") as there is no global information to make use of.

An example of a P2P system for a more benign environment is BitVault. It is primarily concerned with management and availability of data and is characterised by a massively parallel repair functionality. Unfortunately, it assumes a completely benign environment: all resources are fully controlled in a uniform environment, making it applicable to a data centre environment rather than a wide-area system.

Finally, the work presented on replica placement algorithms is geared towards more unstructured and dynamic environments. It often suffers from practical limitations that affect the viability of the proposed solution. For instance, the work by [49] requires knowing the overall popularity of a data item; in reality, this can be very difficult. It also does not solve the problem of where to place replicas, giving only the best replication factor. [159] addresses most of these issues in great detail and also tackles implementation issues. Nonetheless, it continues to aim at more dynamic environments. Its applicability to situations with a reduced number of nodes but comparatively very large number of files per node is also not obvious: but this has not been the goal for their system. As such, if implemented in a more benign environment, it is likely that the resource usage would not be ideal.

2.3 Grid Data Management

Grid Computing [74] is a distributed computing paradigm that proposes to aggregate geographically distributed and heterogeneous resources to provide a unified, secure and pervasive access to their combined capabilities. As a result, Grid applications can take advantage of many networked resources and distribute their usage across a large infrastructure.

Grid computing is therefore an appropriate paradigm for handling very large datasets in heterogeneous environments. This follows naturally from its initial intent, which is to focus on large scale resource sharing. In this section, the relevant work in the Grid Computing domain is introduced.

The Grid Computing community has created a set of technologies to address many of the associated challenges. These include authentication, authorisation, resource access and resource discovery among dynamic collections of individuals [77]. These dynamic collections are called Virtual Organisations (VOs).

A VO defines the resources available to its users and their usage conditions. A VO also provides protocols and mechanisms for applications to determine the suitability and accessibility of available resources. In practical terms a VO is defined using mechanisms such as Certificate Authorities (CAs) and trust chains for security and is implemented through mechanisms such as GSI [76].

Grids are usually constructed using a service oriented paradigm. This began with the definition of an open grid services architecture [75] and infrastructure [167], which were later superseded by the Web Services Resource Framework (WSRF) specification [72]. In WSRF, resources are offered through the invocation of Web services with enriched functionality such as life cycle and state management.

An example of a Grid service is the OGSA-based Data Access and Integration (OGSA-DAI) [10]. OGSA-DAI is a middleware product that allows data resources, such as relational or XML databases, to be accessed, integrated and federated via web services. OGSA-DAI aims to hide the lower level data handling issues (e.g. file replication) from the users, enabling users to depend only on integrated database technologies. Some work remains though, to have OGSA-DAI services operating at the required petabyte scale.

2.3.1 Data Grids

One of the early Grid Computing contributions to the data management problem is GASS or "Global Access to Secondary Storage" [23]. It consists of a system designed to manage secondary caches, which can be seen as a logical evolution of the client-side caches built into distributed file systems such as AFS. GASS claims to support bandwidth management rather than latency management of distributed file systems, but its functionality is very limited.

Based on the experience of GASS and the Grid Computing paradigm, Data Grids [43] were defined as specialised Grid architectures designed to handle distributed management and analysis of large datasets. These architectures are loosely defined to accommodate various models of operation and are tightly integrated with "Grid dynamics": security, awareness of virtual organisations and access to fast-changing large sets of resources.

The "Data Grid architecture" consists of two main components: one responsible for storing and retrieving data and another for bookkeeping. [43] also introduces higher-level services to integrate all the individual lower-level services into a coherent set, defining a Replica Management service, capable of moving files between Grid centres and doing all the necessary bookkeeping. In addition, it also defines the Replica Selection and Filtering service that would decide on-demand replication.

An early attempt to implement the Data Grid architecture was done by Reptor [104], a prototype implementation whose scalability has not been thoroughly evaluated. On the bookkeeping aspect, "Giggle" [42] is the reference work. Giggle consists of catalogues mapping logical names to physical replicas so that users can reference data by a logical name independently of its physical location. These catalogues can be layered similarly to LDAP. Not surprisingly, as the scale increases, there have been proposals to move to a P2P-based approach for searching data, based on distributed hash-tables [35]. In fact, such convergence between P2P and Grid Computing has been defended by the Grid authors as a way to tackle increasingly complex problems [73].

[169] has proposed a taxonomy for the organisation of Data Grids. These can be *monadic*, where all data is aggregated in a central repository that users contact to retrieve data; *hierarchical*, where there is a single source of data distributed in a tree-like form across different resources; *federated*, a model first proposed by [139] and based on distributed databases where each resource owner maintains ownership of its data but shares it with others; or *hybrid*, which is a combination of the other models.

To handle file transfers in a Data Grid, GridFTP [5] is the most commonly used protocol. GridFTP is a set of extensions to the File Transfer Protocol (FTP) that define a general-purpose mechanism for secure, reliable, high-performance data movement. GridFTP functionality continues to evolve with e.g. support for transfer stripping [6] or pipelining of small files [32], with the goal of increased performance for large scale data movement. In [101], a detailed analysis of GridFTP use is made, which confirms its widespread popularity.

Influenced by the Data Grid principles, work started at CERN on GDMP (Grid Data Mirroring Package) [150]. The goal was to provide a mechanism to transfer Objectivity database files between computing centres on the Grid. This work was later expanded [160] to become a generic file and object replication tool. It introduced the concept of a storage system subscribing to collections of files in a producer-consumer model. Files were moved using GridFTP. GDMP was envisaged as a limited prototype system for file movement and its scalability was not investigated.

A different concept is introduced in Stork [100]. The goal is to "make data placement activities first class citizens in the Grid just like the computational jobs", where data placement tasks can be queued, scheduled, monitored, managed, check-pointed, and more importantly completed without human interaction. Stork is a scheduler that defines semantics suitable for data placement activities (e.g. "transfer", "release" or "reserve" jobs). The Stork scheduler then integrates the data placement jobs with the corresponding computational jobs, leading to better coordination between CPU and I/O activities.

More recently, there have been several important examples of integrated replica management services. One such system is by Lamehamedi [107]. It consists of a P2P-based system for replica location and an "intelligent" framework for replication based on user demand and calculations of replication cost. This paper, despite being one of the most comprehensive approaches to managing large datasets on the Grid to date, stills does not address real-life problems such as bandwidth management and does not address issues such as replica consistency or support for tertiary storages, which were modelled as arbitrary file access penalties.

Another recent and thorough attempt to provide more sophisticated Data Grid functionality is detailed in [44]. The system, called Data Replication Service (DRS) is built on top of Grid services such as the replica location services, GridFTP and the Globus Reliable File Transfer (RFT) service [124] (which is a wrapper around GridFTP to increase its reliability). DRS, which uses WSRF, is based on a pull model for data replication, and implements a distributed metadata architecture, an interface to local storage systems and a file transfer validation component. It uses a plug-in approach for custom deployment of data centre-specific configurations.

Despite the examples of Data Grid contributions, it is clear that many important challenges remain to be addressed. Recently, in [58] some of these open issues are discussed with some insight given on the interactions between workflow execution and data management.

2.3.2 Storage Systems

Several storage systems have been developed in the context of Data Grids. These are typically middleware built on top of existing file systems, which are linked together into a coherent set using Grid technologies. In this section, I present the most relevant work in this context.

The Storage Resource Broker (SRB) [17] is a middleware that aims to provide a unified view of the data files stored in disparate media (disks, tape archives or databases) and locations. It provides the capability to organise its data into virtual collections independent of their physical location and organization.

An SRB installation follows a three-tier architecture, where the bottom tier is the storage resource, the middleware lies in between and at the top is the Application Programming Interface (API) and the metadata catalogue (MCAT). SRB hosts collectively form a Data Grid where each server implements trust virtualisation using mechanisms such as GSI that allow SRB servers to share data on behalf of the user.

SRB supports the standard set of POSIX functionality but has been extended with multiple APIs particularly for bulk request manipulation. One of the difficulties in SRB is managing the underlying heterogeneous environment. For this it implements consistency checks and multiple reliability mechanisms. This underlying complexity has nonetheless motivated the authors to start a new project, called iRODS [140], which implements management policies as rules that control the execution of remote micro-services, while managing persistent-state information that can be used to validate management assertions. This can be used by end-users to build additional functionality onto iRODS, instead of layering directly into the system as was the case with SRB.

Whilst SRB provides an interesting approach at tackling the heterogeneity of the environment, which is demonstrated by its use by several applications, it does not directly address a data-intensive environment. A data-intensive environment is characterized not only by hosting large amounts of data but also by the need to access and manipulate these large amounts of data near the limits of the available resources. Much of the mechanisms to regulate transfers for replica placement in SRB provide only low-level mechanisms for this tuning.

A very different approach at tackling the heterogeneity problem is adopted by SRM, or Storage Resource Manager [157]. SRM is a specification that can be implemented by existing storage systems to expose their data onto a Data Grid. SRM defines a set of functionalities such as “prepare to get a file” or “move a file to the top storage hierarchy”. The SRM specification is designed to accommodate multiple underlying storage systems with storage hierarchies, such as tape systems and various levels of disk buffers (hence the need for a “prepare to get” prior to a download, bringing the file first to the correct storage hierarchy layer). Various storage systems provide an SRM interface. Examples are the CERN Advanced Storage Manager, CASTOR [16], dCache [79] and the LCG Disk Pool Manager[3] (DPM).

Another example of a Data Grid storage system is Gfarm [166]. Gfarm aims to provide transparent access to remote files in a Data Grid, via a POSIX compliant interface. It aims to scale to tens of thousands of nodes but no such studies have yet been conducted.

2.3.3 Replica Placement

The Grid Computing literature includes a wide range of work on replica placement in the context of Data Grids. In this section, I introduce the initial motivation to the replica placement problem, which is the job scheduling problem on the Grid. I then discuss multiple replication strategies, which are classified into early work, economic-based, quorum-based and cost-based strategies. All these strategies require global knowledge. This is followed by a discussion of replication strategies that require only local knowledge. I then discuss other replica placement work that tackles closely related problems like maximising availability or ensuring a well defined quality of service. Finally, there is a discussion on the importance of file grouping, workload analysis and file transfer optimisation.

In [142] a comparison is made between scheduling strategies that take into account replication and those that do not. The authors conclude that "clearly dynamic replication helps to reduce hotspots created by popular data and enables load sharing" and "if data locality issues are not considered, even the best scheduling algorithms fall prey to data transfer bottlenecks". The conclusion also states that decoupling job scheduling and data replication makes the Data Grid simpler, permits highly decentralised implementations and actually achieves better performance.

In [143] the same authors build upon previous work with more realistic models and also conclude that correlations in file usage could help design better replication strategies. In [123] the authors study "intelligent" staging in a Data Grid. They propose three techniques: detecting sets of files that are needed by other jobs and stage them together and quickly; use time-to-replication to compute scheduling metrics; overlap the execution of data staging and compute bound tasks. The authors claim that while data staging tuning is important, overall performance can be improved by taking into account job expectations and scheduling jobs that stage data with CPU-intensive jobs. This allows a reduction in the overall waiting time.

More recently, [138] followed the same principles by introducing a "data diffusion" technique, which is the ability to provision resources dynamically, handle caches and implement data-aware scheduling. A different view to the scheduling and replica placement

[3] Refer to *https://twiki.cern.ch/twiki/bin/view/LCG/DpmGeneralDescription.*

problem is given by [168], where the authors assume data is pre-placed and create a heuristic-based scheduling algorithm to schedule the computation.

The work described above has shown the many interconnections between replica placement and job scheduling but also demonstrated the possibility to decouple the two. Based on these principles, many replica placement techniques have been developed. An early example is [141], where the authors define a model with Grid centres organised onto trees and create six different replication strategies: no replication or no caching; best client (create replica on client that generated the most requests); cascading replication (once a threshold is passed, replicate data to lower level in the tree); caching (client always caches data but as files are large and space is small there is a large cache turnover); caching and cascading (combines both); FastSpread (replicate data along the request path). They conclude that if user requests are random, the best strategy is FastSpread. If there is enough geographical locality on the requests then cascading performs best. Finally, depending on whether the application requires lower response times or lesser bandwidth it could opt for FastSpread instead of cascading, as the latter has lower response times but requires more bandwidth while FastSpread performs reasonably with lesser bandwidth.

Following from this initial work, [37] and [21] introduce an economic model for replica placement. This is based on Vickrey auctions done by each storage system. Vickrey auctions are second-price sealed-bid single round auctions: there is a single bid round where each bidder does not see the bids from others and whoever wins pays the price of the second highest bid. As such, bidders have no interest in manipulating prices. [36] provides a formal evaluation of the economic model using Petri nets.

One of the main goals of replica placement is increased availability. Nonetheless, as previously discussed, this goal can conflict with the need for consistency on the Grid. Quorum-based protocols address this problem by guaranteeing consistent information while trying to maximise performance. In [62] data centres are structured as a three dimensional grid structure. Given N copies of a data object, the proposed protocol logically organises the N copies into a box-shaped structure with four planes. Reads require any pair of nodes on the hypotenuse to agree. Any pair is sufficient which increases fault tolerance. Writes require any hypotenuse copies and all vertices copies to be updated.

In [105] yet another replica placement algorithm is introduced, which is based on cost estimation: as users request data to be copied to a data centre, the system decides to read it directly or to replicate it. The cost function takes into account link bandwidth as well as read and write request rates, where adding a replica decreases read cost but increases write cost. A run time system compares the replication gains to the replication costs and then decides whether to create a new replica. In [106] more details are given on the associated replica placement simulator, including the impact on the algorithm by having node additions and removals (in a manner very similar to the churn effect of P2P systems).

One problem with the work mentioned so far is that it requires global knowledge to make a replica placement decision. While these algorithms could potentially attain a global optimum by using global information, it would be very hard to implement in practice. Also, being centralised, their scalability can be questioned. In [178] a decentralised algorithm is proposed. This relies on a set of heuristics for decision making. One of the caveats is that this algorithm only works for an hierarchical (tree-based) topology.

In [114] and [113] a different goal (and algorithm) is proposed. The goal is to maximise data availability and for this two new metrics are introduced. These are the "System File Missing Rate" (the number of files potentially unavailable) and the "System Bytes Missing Rate" (the number of bytes potentially unavailable). The proposed algorithm is greedy and relies on a prediction function.

[116] and [118] study the problem of optimal placement of replicas on a hierarchical Grid. The goal is to balance the workload on all the available storages. These contributions, which include provable behaviour, make nonetheless rather simplistic assumptions about workload modelling and quality of service: e.g. quality of service is defined by how many nodes can be queried to find a data item.

Essentially all of the work presented relies on simulations and makes a set of assumptions about both the infrastructure and the workload. In [133] more accurate models are proposed for storage systems in the Grid. In [101] a detailed analysis of one year and a half of GridFTP transfer traces is introduced. The authors come to some surprising conclusions such as that most transfers are actually for small data files (the intuition was that mostly large data files were used). They also confirmed some known assumptions that stated that the network resource provisioning in a Grid is typically good and most transfers are routed through fast links.

Another example of Data Grid work based on real workloads is presented in [64], where an analysis of real experiment traces (from the High Energy Physics DZero experiment) is done. This analysis is based on 27 months of running. The authors discover that the file sizes are arbitrary and mostly limited by the maximum size supported by the underlying storage system. They also show that the scientific community data is more uniformly popular than Web data, which is known to follow a Zipf-like distribution [31]. This contradicts the models used in many of the previous contributions. The authors also show the emergence of groupings of files they call "filecules". Filecules were initially presented in [1], and are defined as disjoint groups of files characterized by having simultaneous access. The authors then develop a set of scheduling algorithms based on this concept and show it outperforms other alternatives. Future work aims to continue this work by exploiting correlations between "filecules".

Previous analysis also showed some limitations with the GridFTP protocol, which is the most widely used for Data Grid transfers. In [177] a technique is proposed to improve transfer times in GridFTP by using multiple parallel source storages at the same time. Recently, in [94] a more elaborate technique is proposed that employs adaptive replica selection to transfer different chunks of the same file by taking into account dynamically changing network bandwidth. This work has lead to the emerge of GridFTP overlay networks in [96] and [95]. The authors propose two optimisation strategies to improve the performance of data transfers over shared public networks: multi-hop path splitting and multi-pathing. Multi-hop path splitting consists of replacing direct TCP connections between source and destination by a multi-hop chain. It is well known that shorter TCP connections (with more hops) have advantages as the network round-trip time is shorter and packet loss more quickly detected. Multi-pathing consists of stripping data into chunks and sending disjoint pieces through multiple overlay networks in parallel. In addition, the authors propose a path determination heuristic and a file transfer scheduling heuristic for batch transfers.

2.3.4 Discussion

Grid Computing aims at creating a set of technologies that enable large, dynamic organisations to operate in distributed environments. This is accomplished with the integration of distributed resources by a set of middleware services, with the goal of providing a coherent interface. The applications that motivated the emergence of Grid Computing - and the Data Grid architecture in particular - are the same large scale applications addressed in this work. Therefore, Grid Computing and the Data Grid architecture are of special importance to my own proposed solution in later chapters, which can be classified as an instantiation and extension of the Data Grid architecture.

The Data Grid architecture defines multiple, decoupled services to handle management of data. It focuses especially on the replica placement and bookkeeping services. Nonetheless, no implementation currently exists that provides more advanced data management capabilities in a satisfactory manner. For instance, DRS [44] provides higher level tools to manage data replication. Nonetheless, it does not include mechanisms to automatically handle data loss and recovery. Instead, following the Data Grid architecture, these are left as separate services to be coupled at the application level.

Grid storage systems can be split into those that provide only a thin interface (in the case of SRM) or a thicker middleware (e.g. SRB). Nonetheless, none of the proposed systems is shown in the literature to operate across multiple data centres and at the petabyte scale. However, the manner by which these systems are integrated with lower-level middleware is noteworthy, not only for the resulting uniform interface but also for the many difficulties authors faced in maintaining the consistency of data.

Explorations into the Data Grid architecture quickly came to the conclusion that a separation of concerns is desirable between the scheduling and replica placement problems. As a result, many authors proposed several replica placement models. Unfortunately, none of these contributions meet simultaneously the following three goals: to be based on realistic assumptions of the underlying infrastructure (e.g. to include network latency and parallelism considerations in the transfer layer); to be evaluated against realistic workloads (e.g. not assuming Zipf distributions for data access requests); and to encompass more advanced quality of service guarantees (e.g. priorities in the transfers).

Finally, it is important to note that many contributions signal the use of increasingly complex and configurable policies (iRODS), more advanced data organisation techniques ("filecules"), near optimal replica placement techniques as well as the optimisation of the lower level file transfer layers (GridFTP overlay networks). These are goals important to this work.

2.4 Cloud Data Management

Cloud Computing is a recent distributed system paradigm, which has been loosely defined as "a nexus of hardware, software, data and people which provides online services" [67]. This designation encompasses many different aspects. In this section I start by describing existing clouds and then move on to specific technologies, which are divided by file-based storage systems, structured data storage systems, and programming systems for the cloud.

Clouds are designed to make scalable computing easier. This is achieved by a combination of both infrastructure and software services [174]. For a start, clouds hide the data centre operations away from the application developers. Cloud providers achieve economies of scale by building their own homogeneous data centre infrastructures. Users and developers then build and host diverse systems on top of these homogeneous infrastructures by using virtualisation. Because clouds are provided by private entities, they are based on the Utility Grid concept[4], re-implementing the old business model of paid, time-shared, resource usage.

An example is Amazon's Elastic Computing Cloud[5] (EC2). EC2 allows users to create machine images. These are then uploaded and executed on a set of distributed resources hosted by Amazon, while giving the user many operating options, including the choice of operating systems and static IP addresses, etc.

One disadvantage with cloud computing is the lack of user control [86] since the infrastructure (unlike desktop PCs or local clusters) is not owned by the users. Data

[4] Not to be confused with the Grid Computing concept. The Utility Grid refers to the electrical power grid.

[5] Refer to *http://aws.amazon.com/ec2/*.

ownership issues may arise when a cloud provider goes down or disappears. Also, the development environments are neither uniform across clouds nor very rich when compared to traditional development environments.

2.4.1 File-based Storage Systems

In addition to the EC2 service, Amazon provides a storage service called Simple Storage Service[6] (S3). S3 is a commodity-priced storage utility, which provides web service interfaces to store and retrieve data. It is a fundamentally different approach to a distributed file system as it encompasses both the hardware and software layers.

S3 is primarily designed for storing large data objects. Independent studies [134] have confirmed this, with small objects suffering from transaction overhead. There are other interesting observations such as the metadata bookkeeping in S3, the write-sharing semantics and the internal data placement strategies.

S3 stores objects. Each object is identified by a unique key and may have a custom set of metadata assigned as key/value pairs. Objects are organised into "buckets". A single user account is allowed 100 buckets in a shared namespace. Although the motivation for this design feature is not documented, it is probably to prevent abuses of the namespace.

Whenever data is stored on S3, it may not be visible until the changes are fully "propagated". Although no working details are given in the available documentation, it is likely that internally S3 makes multiple copies of the data items. Similarly, when deleting data, the objects may be visible whilst the deletion is not propagated.

In [134] the authors also noticed different data access performance depending on the downloading node, leading to the authors' suspicion that S3 operates multiple data centres and makes data placement decisions depending on the location of the user creating the bucket that stores the data. Also, S3 supports BitTorrent as the download protocol, allowing for substantial bandwidth savings if multiple concurrent clients demand the same set of objects.

Several studies (e.g. [60]) have made a case for the applicability of S3 for a scientific environment, characterized by dynamic communities and potentially large amounts of data. [59] is one such detailed study, which concludes that for a data-intensive application with a small computational granularity the storage costs are insignificant as compared to the CPU costs, in which case cloud computing offers a cost-effective solution.

While S3 provides a very good service with a low number of failures [134], it is not adequate for large collaborations. One limitation is its simple security model that is inadequate to manage large, dynamic, sets of groups. Also, the authors argue that

[6] Refer to *http://aws.amazon.com/s3/*.

the S3 storage model should allow for the un-bundling of its infinite durability, high availability and fast access into independent components with different cost-models, making the service more attractive for large users.

2.4.2 Structured Data Storage Systems

In this section, I describe two cloud-based systems that support structured data, with the goal of understanding the internal design decisions, in particular concerning scalability.

Google's BigTable [41] resembles at first a database but has a significantly different interface. There is no support for the full relational model and transactions are limited to single rows. Instead, BigTable provides a simple data model with dynamic control over data layout and format. An important aspect is that clients can reason over the locality of data. BigTable is designed to cope with both throughput oriented workloads and latency-sensitive workloads and is reported to store several hundreds of terabytes of data. The implementation makes use of the Google File System (GFS) for storing persistent data; it also uses Chubby [33], a distributed locking service developed by Google.

While BigTable provides an interesting and very scalable system for storing very large datasets, it is still designed for a single data centre. There is ongoing work on building cross-replicated BigTables with multiple master replicas but no details are available yet.

Another system that handles structured data storage, but across multiple data centres is Amazon's Dynamo [57]. This is a key/value storage system whose goal is to provide a very high level of availability. Dynamo implements an eventually consistent [170] data store: that is, all updates reach all replicas, eventually. Dynamo internally places replicas using a variation of consistency hashing combined with a quorum-like technique. Users can specify the minimum number of consistent reads (R) or writes (W) in different nodes, out of N nodes, before the operation is accepted. This allows users to configure the availability, durability and consistency properties of their stored data. It also uses a gossip-based mechanism for failure detection and a membership protocol, somewhat similar to P2P systems.

Because the goal is high availability, it may happen that multiple versions of an object are created. Vector clocks [108] are used so that clients have information on eventual concurrency and can act on it. Finally, it can temporarily place objects in another node if the recipient is temporarily unavailable (called hinted handoff), and makes use of Merkle trees [127] for replica synchronization of multiple object versions.

Dynamo has been designed for a non-hostile environment and has no built-in security. Although it replicates data across data centres, it assumes all connections are over a fast and dedicated network. In addition, it targets mostly small objects. It is likely that this replication would be significantly more complicated if Dynamo had to manage much larger data objects.

In the following section, I briefly describe how these systems are integrated with higher-level abstractions, which are then used for data analysis. The creation of new data analysis paradigms is of special interest because they provide novel mechanisms to perform ad-hoc analysis over extremely large datasets.

2.4.3 Cloud Programming

MapReduce [55] is a programming model and associated implementation for processing and generating large datasets. Users specify computation in terms of map and reduce tasks, inspired by the primitives present in Lisp. There is an underlying system that takes care of computation aspects across large clusters of machines, including making tasks run in parallel, handling failures and scheduling intra-machine communication. MapReduce is reported to process up to 20 petabytes of data per day [56]. The implementation is geared towards large scale clusters of commodity PCs connected by Gigabit Ethernet and uses the Google File System for permanent storage.

There are various improvements proposed to the basic MapReduce model. [119] expands the MapReduce abstraction with a wide-scale distributed stream processor called Mortar. The stream processor manages the placement and physical data flow of the operators across the wide area. [45] implemented a new Merge step so that the support for multiple heterogeneous datasets would be easier. Pig Latin [132] goes even further developing a new language somewhat in the style of SQL but implementing a low-level procedural style. It is an interesting system, for read-only analysis, as it gives users detailed control over the query optimization. This is done by having users define explicitly each step of the data transformation. According to the authors, this property is critically for scalability. Pig Latin also builds on top of Hadoop[7], an open-source implementation of MapReduce.

2.4.4 Discussion

The recent emergence of successful cloud computing projects is likely due to the innovative view taken at solving the data management problem. First, clouds facilitate the maintenance of large computing farms. Most important though are the interfaces exposed to developers. Cloud computing has succeeded in creating a synthesis between database and file systems. Many authors (e.g. [84]) have argued that this is an essential condition for petabyte-scale archives. The new programming approaches (or rather, the

[7] Refer to *http://hadoop.apache.org/core/*.

innovative reuse of old primitives) are also providing additional flexibility to the design of scalable, parallel applications.

Nonetheless, no cloud system currently includes techniques for managing large amounts of wide-area distributed data over heterogeneous infrastructures, although all identify this as future work. Amazon recently launched a new service (Amazon CloudFront[8]) aimed at reducing latency for accessing S3 from various locations. It is also interesting to note the trend of allowing users to make trade-offs between availability, durability and consistency in the data store. Amazon's S3 is also interesting for some of its design considerations, such as the simplistic approach of imposing user limits (e.g. every user can have only 100 buckets). These simplifications are necessary for scalable systems. Amazon's Dynamo and Google's BigTable have also achieved greater scalability than parallel databases at the cost of avoiding complex transaction support but requiring customisation of the analysis code.

The issues with Cloud Computing are somewhat symmetrical to those of Grid Computing. While clouds have been designed from "top-to-bottom" to provide specific capabilities, Grids have been designed to accommodate heterogeneous resources with richer interfaces at the higher layers. As the usage of clouds increases, users will face issues tackled within the Grid Computing domain, such as managing large and dynamic collections of individuals, associated security issues and the ability to build applications that span multiple clouds. None of these issues is currently addressed [68].

2.5 Scientific Applications

There are several examples of scientific applications with demanding data handling requirements. This section describes the systems that have been built in the context of these applications.

The Sloan Digital Sky Survey (SDSS) project provides Internet access to its data [164]. Originally the system stored the data using the Objectivity/DB[9], but the data store was later moved to a relational schema and uses the Microsoft SQL Server[10]. The system holds several terabytes of data and is reported to maintain fast user response to queries [164]. According to the authors this is achieved using a combination of physical database design decisions and SQL Server functionality. This includes the spreading of database files across several volumes and the ability to stripe tables across files and disks along with pre-fetching functionality. This application is an important example of relational database scalability achieving the terabyte scale, but is not a wide-area distributed system.

[8] Refer to *http://aws.amazon.com/cloudfront/*.

[9] Refer to *http://www.objectivity.com*.

[10] Refer to *http://www.microsoft.com/sql/default.mspx*.

The Laser Interferometer Gravitational Wave Observatory (LIGO) project has built the Lightweight Data Replicator (LDR) system [135]. The capabilities provided by LDR have since been added to a more sophisticated Data Replication Service [44] already described in Section 2.3.1.

The PhEDEx project [145] has been developed in the context of the CMS experiment at CERN. It provides a data distribution system based on a series of collaborating agents. The agents are persistent, stateless processes that manage specific parts of file replication at each storage in the network. It is the system that presents most similarities with the work presented in this thesis, particular in regards to the data distribution "axis" discussed in Chapter 1. (In fact, PhEDEX has been developed in parallel to the work reported in this thesis, since both projects are part of the Large Hadron Collider at CERN.) Important differences concern the strict separation in PhEDEx between the bookkeeping component and the data distribution functionality. PhEDEX is primarily concerned with managing the distribution of data in the wide-area network, while the system presented in this thesis is also concerned with handling end-user bookkeeping requirements.

Following on the recognition that scientists are designing their own specialised user interfaces and tools to marshal digital resources, several projects develop application-oriented interfaces for the scientific community. Some examples are given in [176] as part of the TeraGrid project, where researchers develop Web applications or portals that attempt to hide many of the lower level complexity from the end-users.

2.6 Discussion

None of the systems described in this chapter matches the needs and environment constraints addressed in this thesis. This includes the ability to manage very large sets of data, at the petabyte-scale, across multiple data centres on the wide-area network while providing rich user functionality. Nonetheless, most active projects identify this as an area of future work (e.g. GPFS, Lustre and more recently GFS and Amazon CloudFront). The Data Grid architecture is the closest applicable distributed computing paradigm, but none of the reviewed Data Grid implementations provide sufficiently rich functionality, but instead propose decoupled building blocks for developers to built application-specific systems.

It is nonetheless instructive to highlight some architectural design decisions adopted in these systems. For instance, metadata is most often handled by a separate service (e.g. Lustre, GFS or the Data Grid architecture). Most distributed file systems use a single central metadata service with some form of built-in redundancy (e.g. GFS, Lustre). Other systems, such as P2P systems, apply partitioning and searching techniques directly in the overlay network for increased scalability, as a mechanism to cope with the wide-area environment.

Another observation is that the more recent systems do not store user files as individual files on the storage. For instance, Lustre, GFS and HDFS split each file into shards. Also, most systems maintain at most POSIX-like semantics and newer systems such as GFS, HDFS and S3 are not POSIX compliant at all, allowing the support for newer and more scalable access interfaces.

Several systems support background replication. For instance, Amazon's Dynamo guarantees that updates are eventually propagated in the background without user intervention. Grid services such as DRS or Stork ensure that user replication requests are processed in the background as well. S3 also supports eventual propagation and does not provide immediate data storage guarantees or transactional behaviour.

The concept of overlay "networks" is also presented in various forms throughout multiple contributions. P2P define overlay networks as a mechanism to introduce some structure into otherwise chaotic systems. Recent proposals to GridFTP also include overlay channels as a mechanism to improve transfer performance. This highlights the usefulness in this concept to provide some form of structure, control and increase the quality of service.

In addition, several contributions exploit internal data organisation properties. For instance, "filecules" are discussed in the context of grid computing while S3 uses "buckets" to organise its namespace and data placement.

Most contributions also acknowledge the need for a close link between the data and its processing, while maintaining some separation of concerns between the two. MapReduce is one such example, with a split between the map and the reduce tasks. Within the grid computing domain, the same principles have been recognised as well (e.g. intelligent "staging" or correlations between file usage and replication strategies).

Finally, many contributions discuss heterogeneity. Data Grids aimed from the start to support heterogeneous environments. This trend is now being adopted by distributed file systems. For instance, HDFS already supports more than one backend. Lustre is working on a Control Panel to support bandwidth management on the WAN, enabling complex setups that span multiple data centres ("heterogeneous" network environment). Ongoing work on cloud computing also focuses on transfers across clouds.

Nonetheless, unlike distributed file systems that require a fairly uniform setup across data centres and complex network configurations, the goal is to develop a data management layer that can scale to hundreds of data centres. But unlike commonly used P2P systems, there are reasonably stable associations between centres and well established security policies. Unlike all systems presented, there is the need to impose global quality of service guarantees based on data properties. As such, the design in Chapter 4 seeks to combine the interesting properties of distributed file systems, P2P, grid and cloud computing.

Chapter 3

System Motivation

There is an emerging class of applications that are characterised by very large data stores and complex methods for analysing and organising their data. In this chapter, I define these new *data-intensive applications*, highlighting the computational, operational and human factors that make such applications hard to develop and operate in a conventional computational fabric. Finally, I describe in detail one such application, the ATLAS Experiment, which has served as the main motivation and source of requirements for this work.

3.1 Data-Intensive Applications

In the past years, several authors have forecast a flood of scientific data (e.g. [88], [84]). This is the result of new instruments with extraordinary precision and improved data quality. The analysis of these data becomes a more difficult endeavour, since there is the need for better computational infrastructures and data processing algorithms, which are able to detect subtle effects in large, complex data samples.

In [88] the authors argue for the creation of types of digital libraries for scientific data with similar management services to conventional digital libraries, in addition to other data-specific services. In [84] similar principles are discussed, with emphasis on the creation of smart notebooks for improved data and data analysis layer. Additionally, the authors argue that *peta-scale science* will require new ways of organising data, and that the simple file storage and transfer "*modus operandi* just will not work for peta-scale datasets."

Scientific experiments, such as the astrophysics experiment LIGO[1] (Laser Interferometer Gravitional Observatory), the bioinformatics Gene Expression Database[2] (GXD), or the

[1] Refer to *http://www.ligo.caltech.edu/*.

[2] Refer to *http://www.informatics.jax.org/mgihome/GXD/aboutGXD.shtml*.

particle physics LHC[3] (Large Hadron Collider) experiments are typical examples of data-intensive applications. Nonetheless, there are many other examples outside the domain of natural sciences; for instance, Google [90] and Yahoo [15] have also developed vast data repositories for hosting their services.

All these data-intensive applications have several features in common, which I now identify. The most defining feature is the need to handle very large volumes of data, which is typically achieved by using distributed data repositories that may ultimately spawn multiple computing centres. Reasons for using more than a single computing centre for storing data include *geo-locality* and *failure avoidance*. Geo-locality is the placement of data closer to its users to reduce the network round-trip times in data access. Failure avoidance is related to the existence of multiple copies of the data to avoid permanent or temporary loss of data or access in the event of a catastrophic failure (e.g. fire) or during routine maintenance (e.g. change in the power supply of a data centre).

Data-intensive applications are also defined by their very large processing tasks, where a single processing task may need to read tens of gigabytes of data to produce a single result. This is typically achieved by implementing a significant degree of parallelism in the computational processing.

Finally, data-intensive applications are the result of large multinational companies or consortiums, typically implemented as distributed collaborations that may include various funding sources. Multiple funding sources may bring about the need to comply with local needs and regulations, which results in a new set of non-functional requirements that must be enforced.

3.2 The ATLAS Experiment

In this section, the *ATLAS Experiment* is described, as it is a prominent example of a data-intensive application. After introducing the ATLAS Experiment and its experimental process, the focus is on describing the mechanisms by which ATLAS organises and uses its underlying computational fabric. The discussion does not address the higher-level services created specifically for analysis because, as shown in later chapters of this thesis, higher-level services are critically dependent on the structure and performance of the middleware layers.

ATLAS is an High Energy Physics (HEP) experiment intended to search for new physics discoveries in the head-on collision of two highly energised proton beams. The experiment is physically situated at CERN, the European Organization for Nuclear Research, on the border of Switzerland and France.

[3] Refer to *http://public.web.cern.ch/public/en/LHC/LHCExperiments-en.html.*

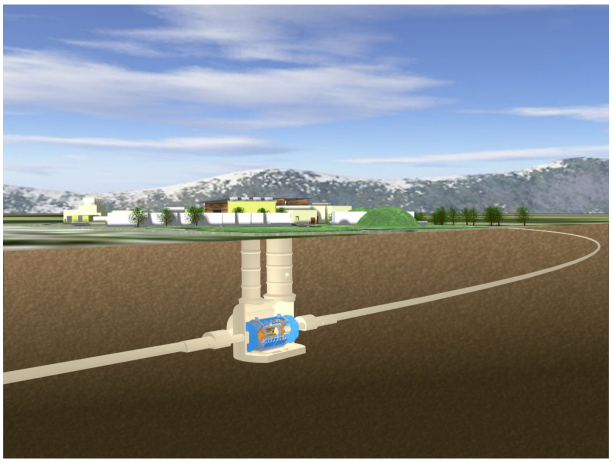


Figure 3.1: A schematic representation of the ATLAS Experiment site. The LHC particle accelerator tunnel is also shown, crossing the ATLAS underground cave.

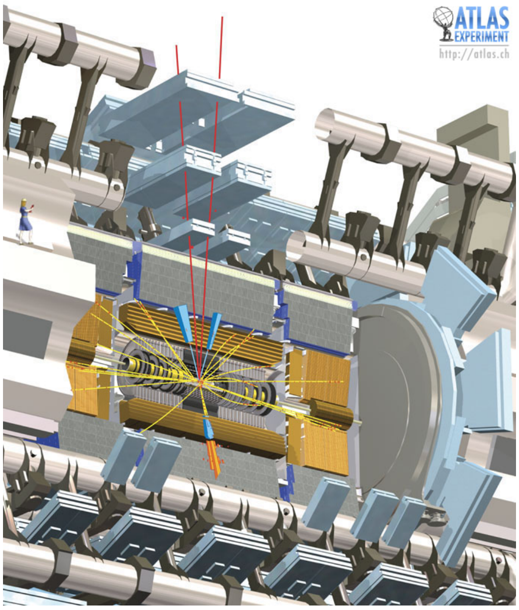


Figure 3.2: Representation of a Higgs Boson event.

The ATLAS Experiment relies on two large devices. One is the LHC particle accelerator, a 27-kilometre ring buried 100 meters underground, where particles are accelerated near the speed of light. The other is the ATLAS detector, a 44 meters long machine with a diameter of 25 meters, weighting over 7000 tonnes, with over 100 million electronic channels and more than 3000 kilometres of cable.

Figure 3.1 illustrates how these two machines operate together. The LHC particle accelerator produces two beams of particles (protons) that circulate in opposite directions. These beams are made to cross each other and collide within the centre of the ATLAS detector. The outcomes of these collisions are detected by the electronics of the ATLAS detector, as shown in Figure 3.2.

Although the ATLAS machinery is physically situated at CERN, the experiment is

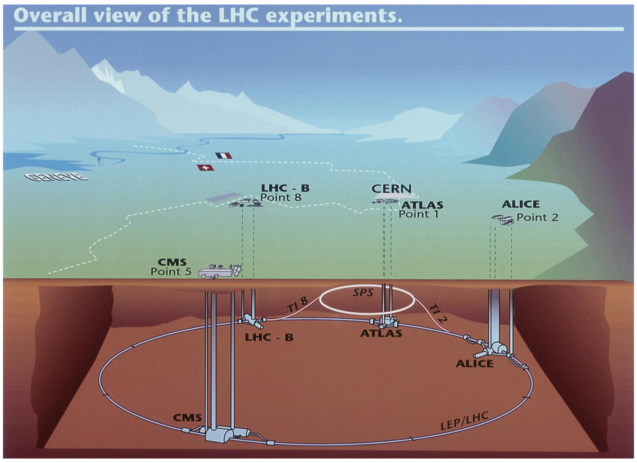


Figure 3.3: The scale of the LHC. The area under which the LHC tunnel can be found is near Geneva and Lac Leman. The French Alps and *Mont Blanc* can also be seen in the background.

the responsibility of a distributed collaboration called the ATLAS Collaboration, which comprises two hundred research institutes and universities worldwide. These centres contribute to the funding of both the ATLAS detector and the LHC accelerator. The total cost of the project is estimated to be as high as €6.4 billion.

The LHC particle accelerator complex is also used by other HEP experiments such as Alice, CMS and LHCb. These are shown in Figure 3.3. Nonetheless, in this section I describe only the computational aspects of the ATLAS Experiment. All these experiments have significant similarities in their computing needs, even though their physics goals vary. Nonetheless, ATLAS presents the highest volume of data, the largest collaboration and hence, the most difficult (and interesting!) computing challenge.

3.2.1 Experimental Process

Particle collisions such as the one previously shown in Figure 3.2, are called *events*. These are detected by multiple layers of detectors (called sub-detectors), which are part of the ATLAS detector. New physics discoveries are obtained by analysing the outcome of these collisions. Physicists hope to understand the fundamental physical processes of nature by detailed analysis of observed events. These events can be, for instance, the resulting trajectories of the debris of a collision.

Observed events are read out from the detector electronics during the collisions. These are written into data files and are generically called *real data*. ATLAS also relies on *simulated data*, which is obtained using Monte Carlo techniques. Monte Carlo simulation is used to compensate for the bias in the machine and understand the observed events: to understand the behaviour of sub-atomic particles it is necessary to understand how these interact and are influenced by the detector itself, i.e. how they interact with the particles that constitute the detector. As such, there are complex software algorithms

Data Type	Size	Unit
RAW Size	1.6	MB
ESD Size	0.5	MB
AOD Size	100	KB
Simulated RAW Size	2.0	MB
Simulated ESD Size	0.5	MB

Table 3.1: ATLAS event data sizes per data type.

that simulate the entire *data taking process*: the LHC accelerator, all of the ATLAS detector and the resulting collisions. The simulated data is then compared with the real data, helping to fine tune the analysis process by removing any bias in the observed events caused by complex interactions of the particles with the ATLAS detector.

The ATLAS Experiment has defined multiple data types for both real and simulated data, of which there are three main types. One is the *RAW* data, which corresponds to the (raw) electronic signals from the detector, or from the simulated version of the detector in the case of simulated data. *ESD*, or Event Summary Data, is a summarised object-oriented representation of the RAW data, intended to make access to RAW data unnecessary for most analysis. *AOD*, or Analysis Object Data, is a reduced event representation more suitable for user analysis. The size required to represent an event is reduced considerably from RAW to ESD to AOD data types, as shown in Table 3.1.

The data processing activities can be classified into *production*, *group* and *user analysis* activities. Production activities require the processing of large amounts of data, which in turn requires large amounts of computing resources. As such, production activities, also called large scale activities, are defined and agreed at the collaboration level. Examples of production activities include the processing of new real RAW data (from the collisions produced in the LHC accelerator by the particle beams) or re-processing of older sets of RAW data with newer algorithms that are of wide interest to the ATLAS physics community.

Physicists that share specific research interests (e.g. Higgs analysis, Muon physics, etc) are organised into groups. These groups engage in medium to large scale activities that are subjected to some central coordination among the various groups. This coordination ensures that, at a high-level, each group is allocated the resources it needs without conflicting with other activities. For instance, the Muon group may temporarily need to obtain very large statistical samples to confirm a result[4], and so requests and negotiates the usage of a larger set of the available computing resources for a period of time.

User analysis are small-scale, decentralised activities that involve running over diverse sets of data: e.g. to test a new Monte Carlo algorithm in a small subset of AOD (real and/or simulated) data to check whether it performs better than previous algorithms.

[4] Note that the Monte Carlo nature of the analysis process may require very large samples to obtain meaningful results.

	Tape (Petabytes)	Disk (Petabytes)
Tier-0	5.7	0.39
Sum of Tier-1s	9.0	14.4
Sum of Tier-2s	0.0	8.7

Table 3.2: ATLAS resource pledges in 2008.

Improvements obtained from user analysis are usually confirmed at the group level and can later be merged into future production activities.

One important observation is that, in practice, most of these activities cannot be predicted in detail. It is not always possible to know in advance how much Monte Carlo data needs to be generated for fine-tuning a specific process; or how long a (real) *data taking run* will take, before some intervention is required in either the accelerator or the detector machines. Real data taking runs can last from several minutes to many hours or days and each run can have significantly different *run conditions*, which may or may not be satisfactory for the desired analysis. Similarly, Monte Carlo analysis may produce surprising results and expose new software bugs when samples are generated at a very large scale. This implies that activities may need to be rescheduled so that any necessary fixes are applied. Other changes can be due to unexpected changes in the run conditions, or due to the identification of promising results, or simply the need to respect physics publication deadlines.

As a result, the ATLAS data processing tends to be a very dynamic and fast changing process, with the relative importance of specific activities changing quickly as the experiment proceeds. Not all activities are centrally coordinated but there is always a degree of negotiation when allocating resources for the different groups and users. These negotiations occur at regular collaboration meetings, but as discussed these allocations can also change over time.

3.2.2 Computing Resources

Because the ATLAS Experiment has no single owner or funding agency, it uses the available computing resources from each participating centre. These distributed computing resources have been brought together under a Grid infrastructure project called the Worldwide LHC (WLCG) Computing Grid[5]. As a result, the WLCG Computing Grid resources are not owned by a single institution or allocated through any single budget. One practical result is that, despite a close collaboration between all participating institutions, there is a degree of heterogeneity in the resource characteristics. This follows from hardware tenders that are separate over time and come from independent budget allocations. In addition, the scale of the resources at each centre can be significantly

[5] Refer to *http://www.cern.ch/lcg.*

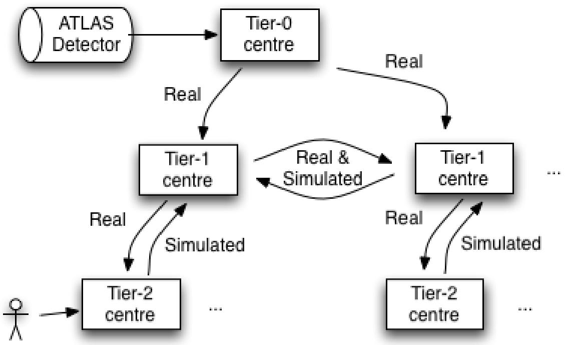


FIGURE 3.4: Simplified overview of the ATLAS data flow.

Activity	Source	Destination	Data Types	Continuous Rate
Real data	ATLAS Detector	Tier-0	RAW	320 MB/s
Real data	Tier-0	Tier-1s	RAW, ESD, AOD	1020 MB/s

TABLE 3.3: ATLAS data export rates.

different, ranging from large, professionally managed computing centres to small-scale university clusters.

ATLAS is not the only experiment that uses the WLCG despite being one of its major users. As such, some coordination is required at a high-level among all major users of the WLCG, to guarantee proper resource allocation for each experiment. This process somewhat mimics the type of negotiation that occurs between different groups in ATLAS, but now among several experiments.

One important consequence of using a shared Grid infrastructure is that the computing resources tend to be used near their maximal capacity, as each experiment attempts to make the maximum use of their allocation.

In addition, because resources are pledged to ATLAS, these pledges come with constraints on how the resources are to be used. For instance, computing centres expect to receive certain percentages of real data as a trade-off for volunteering a subset of their resources for activities that are not of interest to their local users. Since ultimately the success of ATLAS depends significantly in the availability of computing resources, it is of interest to respect the desired constraints and obtain as many resources as possible.

To understand the scale of the ATLAS computing infrastructure, the storage resources pledged to ATLAS in 2008 are shown in Table 3.2. These are being significantly increased as the experiment begins full data taking operations in late 2009.

3.2.3 Data Flow

Having described the goal of the ATLAS Experiment and its computing resources, I now describe how these computing resources are structured to achieve the experimental goals.

ATLAS has divided its participating computing centres, or sites, into multiple layers, based mostly on the quality of service these provide. The Tier-0 layer comprises a single site, which is CERN, where the LHC particle accelerator ring and the ATLAS detector are located. Tier-1 sites (of which there are ten in total for ATLAS) are typically large national laboratories. These correspond to the professionally managed computing centres described in the previous section. Tier-2 sites are mostly smaller centres, typically universities. Each Tier-2 site is associated to a single Tier-1 site. This creates a hierarchical structure of sites for ATLAS.

Each layer has been assigned a different responsibility in ATLAS: the Tier-0 site is mostly concerned with collecting and doing a first processing of new data from the detector; the Tier-1 sites are mostly concerned with reprocessing older sets of data with newer algorithms; the Tier-2 sites are mostly concerned with generating simulated data and are also where most of the users perform their physics analysis.

One of the common misconceptions about the LHC Grid prevalent in the Grid literature, is that the data flow for the LHC is only hierarchical: data is injected at Tier-0 and propagates down to the Tier-1 sites and then to the Tier-2 sites. While this flow is indeed part of the ATLAS data flow, it is only one of the many possible flows. There are other equally important, and significantly harder to manage, data flows. The simulated data (Monte Carlo output produced at the Tier-2 sites) actually goes up and is stored at the Tier-1 sites. This is because the Tier-2 sites do not have tape storage and as such are not considered suitable for long-term data archival, which is a requirement for an experiment such as ATLAS which will run for over 20 years. The output of new reprocessed data is shared among Tier-1 sites and then sent down to Tier-2 sites. Finally, physicists request and share data using any potential site where there is free quota.

All these flows and some of their respective data rates can be seen in Figure 3.4 and Table 3.3. Note that the previous description of the data flow has been significantly simplified, ignoring the various data types and the different data flows for each data type: for a detailed explanation, refer to the ATLAS Computing Model document [2].

The organisation into tiers defines strict roles for each computing site. This implies that some data flows, as shown on Figure 3.4, are statically defined: every data type will be produced and stored in a well-defined subsets of sites. In later chapters, this mode of operation is subjected to further analysis. Nonetheless, I now describe some important non-functional requirements (and one operational reason) that have led ATLAS to adopt this model.

The non-functional requirements are related to resource allocation issues, budgetary constraints as well as the underlying physics interest: resources are pledged to ATLAS but the experiment must try and balance the mismatch between the computing and storage capabilities of different sites with its global needs. Different sites serve multiple experiments, each with its own requirements, so ATLAS has only limited decision power on how resources are allocated overall[6]. Even within ATLAS, different physics groups have diverse needs and interests; nonetheless, many activities are important for the entire collaboration and must be completed, even if those activities are not of immediate interest to any group or centre.

This does not imply that the decision to use static data flows is ideal for maximal resource utilisation. In fact, given the dynamic environment, it is likely not to be the optimal resource allocation. Nonetheless, past experience in ATLAS has demonstrated that dynamic data and job scheduling was severely restricted in practice. One such example was given above, where data produced at a Tier-2 is never kept at that site permanently, but shipped to a Tier-1 site where permanent archival is available. It is instructive to look at this example in more detail.

The production of a Monte Carlo simulation runs at a set of Tier-2 sites where CPU resources are available. The Tier-1 sites cannot do this processing because they are already busy reprocessing older sets of real data; Tier-1 sites have to do this reprocessing of older real data, because no other sites have the required balance between CPU and storage, that is, a significant number of both CPU and storage space (there is more real data than simulated data). In addition, Tier-2 sites tend to have more CPU than storage, making them suitable for CPU intense activities like Monte Carlo simulation; also, as mentioned, Tier-2 sites do not have long-term archival storage (i.e. tape storage) so these cannot hold the final output. In addition, Tier-2 sites usually have limited network connectivity and only a single preferred connection to a specific Tier-1 site; e.g. the UK Tier-2 sites have direct network links to the Rutherford Appleton Laboratory (RAL), which is the UK Tier-1 site for ATLAS.

Therefore, it seems a natural choice to have Monte Carlo tasks run exclusively at the UK Tier-2s (because the Tier-1 is busy) and have their output stored at the UK Tier-1 (where it has permanent storage): there are better network links, and non-negligible human contacts for operational issues, between a Tier-2 site and its close Tier-1. In addition, these tasks may match the research interests of the regional groups. Having tasks run in e.g. a faraway Tier-2 site and then send the output to RAL would certainly complicate operational procedures when problems occur (at the very least due to the different time zones!).

[6] This is perhaps the most striking distinction to a company such as Google, which has also massive computing power available but is able to globally decide how these resources are allocated and guarantee their uniformity, greatly simplifying its own distributed data management needs. Nonetheless, it will be interesting to see, as its own needs grow, if its computing centers will gradually become more and more diverse, leading to the same sort of heterogeneity issues observed for ATLAS.

3.2.4 Usage Scenarios

The previous sections described the experimental process for the ATLAS Experiment, the available computing resources and the associated data flow. To complete this description, this section describes *informally* a fictitious but realistic scenario that illustrates how physicists interact with the data management system in their every day activities.

The ATLAS team based at CERN is working long shifts, attempting to guarantee the conditions required to operate the ATLAS detector. A sub-detector is malfunctioning and this affects the data taking process. The unavailability of a sub-detector causes the stream of data to be different, and some physics groups around the world are particularly interested in confirming recent observations precisely from this sub-detector. The question circulating in the ATLAS control room at CERN and in university meeting rooms worldwide, is whether a new physics effect has been discovered, or whether the results are invalid because of bad sub-detector readings.

The communication between the ATLAS team at CERN and universities worldwide is not always smooth. After all, there are over 200 participating institutes in virtually every time zone. The CERN team decides to run without the sub-detector. The LHC beam will restart soon and the behaviour of the sub-detector is still not understood. Better run with the sub-detector off, to avoid physicists running over potentially invalid data!

Research groups worldwide interested in the physics enabled by this sub-detector are not informed officially of this last moment decision: they will be informed later in weekly operation meetings where the mishaps and successes of the last week are discussed. Instead, they realise the sub-detector must be off: the corresponding datasets are not arriving at their data centre, and the LHC beam should be active at this time of the day.

In parallel, other physics groups continue their activities undisturbed. The sub-detectors they rely on are working properly. Datasets are produced at CERN and sent to interested groups around the world. As data arrives in universities, PhD students start plotting histograms and trying new data cuts to classify the underlying events.

Interestingly, a student working in Orsay for Muon physics finds a suspicious peak in today's data according to his new classification algorithm. He runs the same algorithm over older datasets available in his university's storage system but does not see this effect. He then tries to run the algorithm over an older data sample available in the University of Texas. He is only allowed a very small quota elsewhere, but the University of Texas seems to be rather inactive today and he his given a large resource quota! "I guess they must be depending on some sub-detector that's probably off today...". Interestingly, he sees again the same effect! He then starts browsing the data management system, and checking metadata attributes that describe today's data taking. Today's conditions are different from the conditions of his locally available data, but somewhat similar to what is available in Texas... Nonetheless, there are many other factors to test.

He discusses the issue with his group coordinator. Days later, universities worldwide interested in Muon physics join together for their weekly phone conference. The student describes his findings. The impact is potentially high if confirmed, but more tests are required. The Muon physics group coordinator asks all groups if they are interested in a large-scale Monte Carlo simulation test: generating "fake" (but realistic data), sending it through a software simulation of the detector and running the student's classification algorithm. They agree such test is appropriate: because the student developed a filtering algorithm, large statistics are required to confirm any results.

In the meeting, there is agreement that the Muon physics group coordinator should request a large-scale test. The schedule of this test must be discussed with the remaining ATLAS coordinators, because other physics groups may have similar needs. In particular, this test will require many computing resources: certainly more than what the Muon community itself owns.

Weeks later, the Monte Carlo simulation finally starts. Data centres in universities worldwide are automatically allocated subsets of their computing resources to run this Muon physics Monte Carlo simulation, whose filtering uses the student's new algorithm. Results are sent to the student's own data centre, where additional plots are generated. Clearly, his algorithm is better because the effect also shows in this simulation. More tests are required but if all goes smoothly, his code will eventually be part of an official ATLAS software release. This means that his code will start running directly at CERN in all new data produced at the LHC. In this case, his code enables better classification of events using novel filtering techniques that remove noise from the data more efficiently. Clearly, better filters are very important because they allow new behaviours to be found in the data. On the other hand, some ATLAS coordinators nod their head: if his algorithm really works, then physicists will start asking all last year's data to be reprocessed... and that requires some coordination because the computing resources are not enough for all tasks!

3.3 Summary

This chapter discussed an emerging class of data-intensive applications that are characterised by vast data stores and complex analysis methods. Several important features of these applications were presented. These are the need to handle vast volumes of data, the reliance on distributed data repositories, the need for very large processing tasks that are executed in parallel, and the negotiations that result from having to operate large distributed collaborations with distributed resources.

In addition, the ATLAS Experiment was presented as a prominent example of a data-intensive application. The ATLAS experimental process relies on a mix of real and simulated data. It includes processing activities of varying scales, from large scale production activities, to group and individual user analysis. The experimental process is characterised by a degree of uncertainty, which is a direct result of the complex, underlying physics research.

The ATLAS computing resources are provided by all ATLAS participating centres and used in an opportunistic manner. This results in complex constraints that are negotiated between ATLAS and its resource providers. Respecting these constraints is desirable to ATLAS, to obtain as many computing resources as possible.

The ATLAS data flow was also presented. The ATLAS data flow is a composition of multiple data flows for each of its activities. These data flows are, for the most part, subject to important restrictions. These restrictions are the result of resource allocation and other issues, which condition where data can be produced and where data can stored.

Finally, a fictions but realistic usage scenario was described. This usage scenario illustrates how physicists interact with the data management system in their every day activities.

Chapter 4

System Requirements and Architecture

This chapter introduces design principles for building distributed data management systems for data-intensive applications. These principles follow from a set of requirements and result on the definition of a system architecture.

In the first section I start by introducing the relevant methodology. This is followed by the requirements of each actor in the system along with relevant background information. After introducing the requirements, I introduce and motivate each of the design principles. This is followed by the presentation of the system architecture, which includes the principal entities, system functionality, relevant architectural decisions and the components that form the distributed data management system. The final section includes a discussion on the general applicability of this proposal.

4.1 Methodology

The methodology used in this work is described in Figure 4.1. In the previous chapter, I described the motivation for building a distributed data management system. A set of actors and requirements were identified from this high-level description. These are schematically (and informally) represented in the figure, with each actor connected to a Venn diagram that contains the corresponding set of requirements (represented as dots in the figure). From the requirements, I define a set of general design principles, which are also illustrated in the figure. The design principles are detailed later in this chapter. Finally, I define an architecture that implements the design principles, followed by the system implementation.

The requirements are the result of extensive meetings with the physics community within the ATLAS Experiment. I identified these requirements during formal and informal

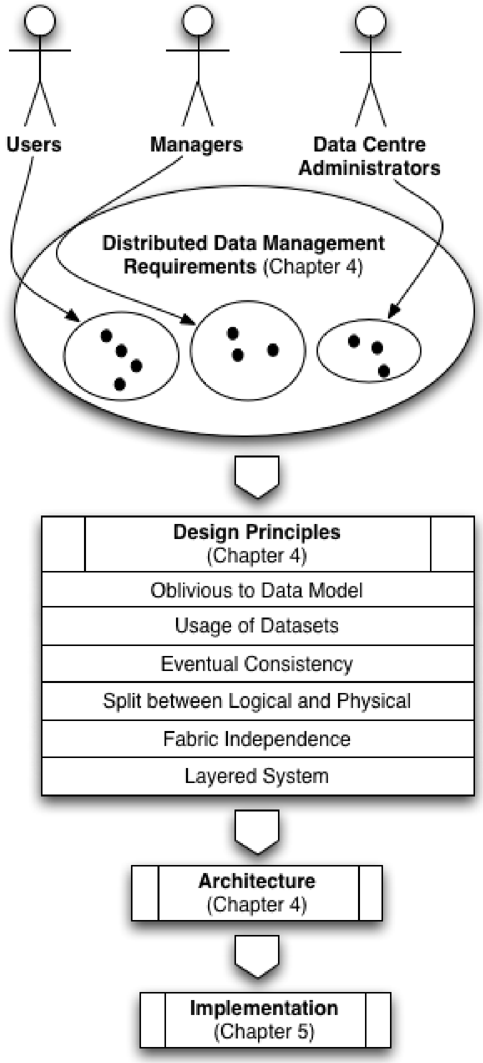

Users
Managers
Data Centre Administrators
Distributed Data Management Requirements (Chapter 4)
Design Principles (Chapter 4)
Oblivious to Data Model
Usage of Datasets
Eventual Consistency
Split between Logical and Physical
Fabric Independence
Layered System
Architecture (Chapter 4)
Implementation (Chapter 5)

FIGURE 4.1: Representation of the methodology.

discussions. These discussions were organised early in this work and involved representatives of various physics communities as well as medium and high-level ATLAS management. The final requirement list was circulated and approved at a later phase by the ATLAS management. In these discussions, I identified three distinct types of actors in the system. These are the *users*, *managers* and *administrators.* (In the following description, a parallel is made where appropriate to the informal description of usage scenarios in Section 3.2.4.)

- **Users.** In the case of ATLAS, the users are the physicists that analyse data and develop new physics algorithms. (In the usage scenario of Section 3.2.4, an user is the PhD student developing new algorithms, or the users expecting to receive data from the sub-detector that had been switch off.)
- **Managers.** The managers are the set of people that are responsible for decision-

making at the collaboration level, e.g. in the case of ATLAS, the managers are responsible for allocating quotas for physics groups. (In the usage scenario of Section 3.2.4, the managers are the ATLAS physics group coordinators and the members of the ATLAS team that decided to switch off a sub-detector.)

- **Administrators.** The administrators are the owners of the computing resources, which may or may not be dedicated to ATLAS (typically the resources are not dedicated). For instance, administrators can be the data centre administrators at each university, which may or may not be part of the physics group.

Care was taken to generalise the process by which requirements were gathered, particularly in what concerns the data centre administrator requirements. Data centres serve multiple communities and their input has been particularly relevant for the non-functional requirements that condition architectural decisions. This is the reason for including a substantial background discussion in Section 4.2.3, on storage technologies and processing facilities, before discussing data centre requirements.

Although this chapter includes a high-level description of the system architecture, the main contribution is the set of design principles. The objective has been to identify, based on the requirements, a set of universal principles that build upon database and distributed computing theory. Database theory has had substantial contributions to data management, and distributed computing is the applicable computing paradigm for this working environment. While the design principles underlie my architectural version of the system, the goal has been to define design principles in a sufficiently universal manner, to accommodate several alternative architectures. In fact, as discussed in the methodology section of Chapter 5, the design has been revised several times, but the essential principles remain unchanged.

4.2 Requirements

I now introduce the main set of requirements for each of the actors.

4.2.1 Users

The main user requirements are:

- “transparent” access to the data, regardless of where it is physically stored;
- ability to store very large sets of data (e.g. terabytes or petabytes of data);

- support for concurrent activities, where concurrent data generation and data processing workflows can simultaneously read and write disjoint segments of a single (large) set of data;
- ability to integrate the distributed data management system with existing data generation and data processing workflows;
- ability to integrate with external metadata[1] tools, so that these tools can reference and annotate the provenance of (stable snapshots of) a set of data;
- ability to maintain private sets of data, which are only visible and accessible to a restricted number of users.

4.2.2 Managers

The manager requirements are concerned with resource allocation issues as well as budgetary constraints and data safety:

- in practice, no single computing centre can afford to host all the data due to the high cost involved and data safety concerns. As such, the system must provide mechanisms for managers to define flows for data access. These flows must enable the continuous and automatic distribution of data among multiple computing centres with minimal human intervention;
- the system must support dynamic priorities for data distribution and gracefully handle a large number of distribution requests;
- the system should adequately support various scales of computational resources, ranging from small clusters in a university department to large scale professionally managed computing centres. It is expected that these resources provide very different qualities of service;
- the system should be minimally intrusive, allowing for opportunistic use of resources. If additional resources can join the system with little or no local customisation, it is likely that there will be more volunteer contributions[2];
- centralised administration requirements should be reduced to a minimum, but mechanisms must be available to impose global policies or perform immediate actions, such as deletion of erroneous data;
- the system must provide an adequate level of monitoring information, both real-time and historical, so that decision makers can analyse past as well as current resource usage and negotiate future allocations;

[1] The word *metadata* is used in this context to refer to simple annotation mechanisms, such as user-defined data attributes based on *(key, value)* pairs.

[2] A clear parallel can be established with P2P systems.

- the system must support the definition of global quotas for users and groups.

4.2.3 Administrators

The data centre administrators have also a set of requirements that must be supported. These requirements are particularly important given that the computational resources are not directly owned by either users or managers. To better understand these requirements, I first present some background information on storage technologies and processing facilities.

4.2.3.1 Storage Technologies

Data storage technologies are concerned with retaining digital data for computing purposes for some interval of time. Traditional categorisation divides these into primary, secondary, tertiary and off-line data storages. The first of these, primary data storage, also known as memory, is directly accessible from the CPU. It is not of concern for this discussion. The remaining storage technologies are relevant, since these form the building blocks for a data centre storage facility.

Secondary storage refers to a type of storage that is not directly connected to the CPU. Instead, it relies on input/output channels and transfers data for CPU processing using an intermediary area of the primary storage. Contrary to primary storage, the secondary storage is not volatile: the data is not lost when the device is powered down. The most widely used example of secondary storage are hard disks; other examples include flash devices (e.g. USB keys), floppy disks, magnetic tapes and rotating optical storage (e.g. CDs and DVDs). Secondary storage is approximately one order of magnitude cheaper than primary storage but has significantly lower access time for reading the data (milliseconds as opposed to nanoseconds).

Tertiary storage is another type of storage that requires an intermediary mounting operation. Before data can be accessed, a robotic mechanism needs to locate the storage device using a catalogue database and physically mount it. The data can only be accessed after this mounting operation is completed. This model allows for extremely large data stores to be accessed without human operation. Nonetheless, the access time is significantly higher than for secondary storage: it is in the order of seconds or minutes as opposed to milliseconds.

Off-line storage, or disconnected storage, is another type of storage that requires a human operator for the mount operation. It is primarily used for long term archiving and information security, since it allows the storage devices to be physically transferred to remote locations.

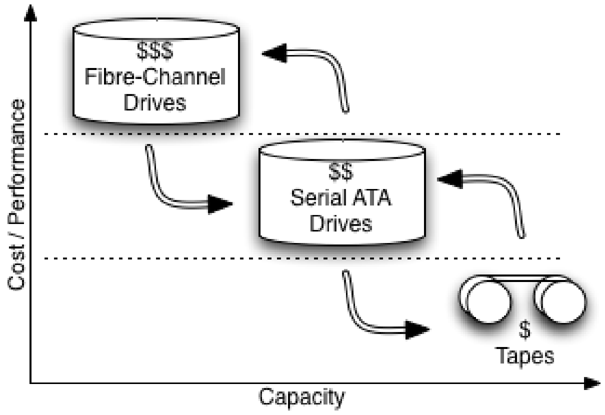


FIGURE 4.2: Representation of a tiered storage.

When large data stores are required, the data is typically stored and shared over a network. That is, direct attached storage is no longer sufficient since it is not feasible to connect all storage devices to a single server. Instead, either storage-area networks or network-attached storage are required.

Storage-area networks (SAN) are an architecture that allows remote storage devices to be connected to a server in such a way that the devices appear as if they are locally attached to the server. That is, the separation of servers from the storage device is done at the lowest possible level in the communication stack: at the block I/O level, which is the building block for a storage I/O. A common reason to use a SAN is when an application requires low-level and direct control over a file system, for reasons of manageability and performance.

Network-attached storage (NAS) differs significantly from a SAN in that the unit being managed are files and not storage blocks. A NAS device is a self-contained computer connected to a network that shares the data files in its operating system using specific client/server protocols. An example of NAS is the sharing of data stored in some computers (the servers) to other computers (the clients) using NFS.

Even though SAN and NAS are distinguished by their working units (block-level versus file-level), the boundaries are being reduced in recent storage offerings that combine both possibilities. This flexibility allows different application requirements to be fulfilled with increased performance.

In addition to SAN and NAS, there are other techniques for designing data centre storage. A commonly used technique consists on breaking down the storage architecture into smaller problems, and then combining different components. These components are divided into tiers, where each tier differs in the type of hardware used, its performance and scale. Each tier is also associated with a specific policy. This approach, which is widely used in data centres, is called a tiered storage model.

An example of tiered storage is given in Figure 4.2. Here, the storage system is composed of expensive fibre channel drivers, less expensive serial ATA drives and even less expensive (according to their capacity) tape drives. Data is moved between tiers as necessary, with less used data moved to lower tiers; in the figure, the lower tiers are composed of serial ATA drivers or tapes.

An extension to the tiered storage model is Hierarchical Storage Management, or HSM. In this data storage technique, data automatically moves between high-cost to low-cost devices, i.e. from devices with less capacity but higher performance to devices with more capacity but lower performance. In a typical HSM scenario, data files which are frequently used are stored on hard disk and migrated to tape if they are not used for a certain period of time. If a user requests a file currently on tape, the file is staged (automatically moved back to a hard disk). If the HSM model is correctly coupled with the application, it is possible that users never notice any actual slowdown because data is preemptively staged. For instance, in a movie streaming service, the first few seconds of a movie are stored on disk and when the movie starts being streamed to a user, the remaining contents are staged from tape.

The HSM model is very popular as it combines high capacity with high performance. The difficulty is in adapting the HSM configuration to different applications so that users are not affected by the slowdown in staging data. In the following section I discuss this subject in the context of processing facilities.

4.2.3.2 Processing Facilities

In this section I describe issues related to the processing of data at a data centre. Although data centres have been discussed informally, a more precise definition follows: a data centre is a collection of computational resources maintained by an enterprise to accomplish needs that go beyond the capacity of a single machine. Among the various types of computational resources that constitute a data centre, the two main resources are the storage and the computing clusters.

Computing clusters are groups of linked computers. They are usually employed for improved performance and availability and are a cost-effective approach compared to purpose-made single computers of comparable performance and availability. Computing clusters are primarily used for intensive computations rather than I/O oriented operations, like those required by a database or a web service. Examples of cluster applications are weather simulation or particle physics event reconstruction.

A factor that distinguishes computing clusters is the degree of coupling between individual nodes. There are two main scenarios. In the first scenario, nodes are tightly-coupled with dedicated network links and extensive communication between nodes. In this scenario, nodes are also often homogeneous. In the second scenario, there is little or no need for intra-node communication. Therefore, it is relatively easier for developers to partition a single large task into sub-tasks (individual jobs). In this scenario, a higher heterogeneity of resources is also common. The applications that can be deployed in this second type of clusters are called *embarrassingly parallel*. In this thesis I focus on data-intensive applications of which the ATLAS Experiment is an example. As discussed in the user requirements, I focus on the sub-set of data-intensive applications that are characterised by a large degree of parallelism in their processing. That is, jobs do not require any intra-job communication during the computation process. Therefore, these are embarrassingly parallel data-intensive applications, each generating and executing many jobs in parallel.

Because a data centre does not serve a single application or a single uniform group of users, a degree of adaptation is required to support different workloads. Even though jobs are relatively small and independent units, they can significantly stress both the computing and storage fabrics. After all, these are data-intensive applications that need to process very large volumes of data.

To cope with diverse sets of applications and users, data centres must partition the available resources to facilitate the management of many different types of jobs. This follows similar principles to those of the HSM architecture, but instead applied to the computing resources. For example, jobs with short expected duration are allocated to a dedicated subset of the computing cluster, typically implemented through separate job queues. Jobs with more demanding primary storage (memory) requirements are also allocated to separate (more expensive) resources.

There is an additional aspect, which only applies to applications that operate in more than one data centre simultaneously. This is the need to distribute data between data centres. Data needs to flow into and out of the data centre. For this, it is typical for data centre administrators to employ data import and export buffers, which are transient storage used to (respectively) write and read data from a data centre storage to an external data centre storage. These additional storage buffers further complicate the definition of an adequate data centre architecture (certainly more difficult than for a single application or for applications homed within a single data centre). It also makes the definition of the distributed data management layer more complex, but it is a necessary condition to ensure that requests from external data centres do not significantly impact local tasks.

Figure 4.3 shows a possible schema where both the storage and the individual computing nodes (called the `CPUs` in the figure) are represented. The storage may be based on SAN and/or NAS and will typically follow, in the most complex cases, a hierarchical storage management architecture. In addition, the computing cluster is partitioned into separate queues: a long jobs queue (e.g. jobs that should take over one hour to finish) and a short jobs queue (e.g. jobs that should finish in a few minutes). There are also import

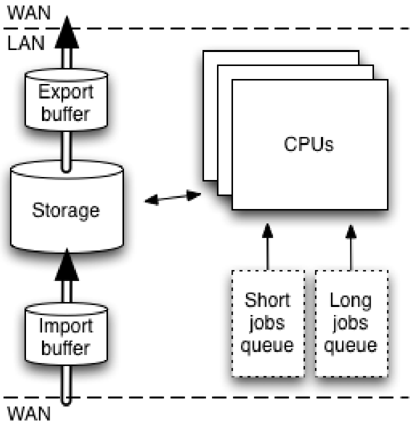


Figure 4.3: Representation of a data centre storage and processing facility.

and export storage buffers, which serve to throttle the rate of data flowing into and out of the data centre.

From this discussion, it should be clear that data centre administrators are presented with a serious design challenge; hence, their requirements are of utmost importance particularly if there is the managerial goal of using many resources in an opportunistic manner. The next section elaborates on this discussion by describing the data centre administrator requirements. These include additional issues such as security, software deployment constraints and data protection.

4.2.3.3 Requirements

The data centre administrator requirements for a distributed data management system are:

- adaptability to the data centre architecture, specifically to the NAS and/or SAN storage, as well as the HSM (e.g. including the magnetic tape storage);
- support fluctuations in the performance of the data centre resources (e.g. storage access may slow down significantly if there is some temporary overload or ongoing maintenance operation);
- ability to operate with minimal information regarding computational and storage architectures, allowing changes to the data centre without extensive system re-configurations;
- ability to throttle inbound and outbound data transfers (outside the data centre), to protect the storage from overload conditions;
- support for both scheduled and unscheduled downtime, with graceful recovery after a downtime (i.e. without overloading newly up servers with requests);

- automated recovery against unexpected data losses, either permanent or temporary (e.g. while a damaged tape is sent for repair and recovery);
- reduced network connectivity requirements from the software stack, regarding both inbound and outbound connectivity, to reduce security risks through outside-world exposure;
- non-intrusive software stack with minimal set of local software and hardware dependencies;
- ease of deployment, including the possibility to perform quick re-installations and re-allocation of any locally installed software.

4.3 Design Principles

In this section, I introduce and motivate the design principles for the distributed data management architecture. These concern the data model and data unit in the system, the consistency model for replicating data, the separation between logical and physical data units, the principle of fabric independence and the definition of an architecture based on a layered system.

4.3.1 Data Model and Unit

The first design principle concerns the data model and data units in the system. A data model is a collection of high-level data description constructs that hide many low-level storage details. Examples of data models from data management theory are the relational data model used in most database systems, the hierarchical model, the network model and the increasingly popular object-oriented and object-relational models.

While many applications have adopted the relational model, this can pose difficulties for some applications. These difficulties are caused by the *impedance mismatch*, as discussed by Gray et al in [84]. The impedance mismatch is the mismatch between the programming model and the underlying capabilities provided by a specific data model.

Many existing applications employ proprietary data models. For instance, the representation of physics events in ATLAS is done using a custom object-oriented model. This presents multiple advantages: for instance, it allows ATLAS to use complex (hierarchical) object references in its event modelling.

Changing an application from an existing, custom-made data model to a generic data model is usually a very difficult task. Even when this adaptation can be done, it rarely utilises the full data model capabilities, because that would require significant changes to large parts of the application code.

As such, the first design principle was not to impose any specific data model to an application that uses the distributed data management system. Applications that are the target of this work have most often developed proprietary data models where data is stored in custom-built file formats. I now highlight this principle.

Design Principle: The distributed data management system is oblivious to the data model used by the underlying application.

If there are proprietary data models (or relational databases, which in turn store their data internally in files), the distributed data management system must manage these files without any knowledge of the data stored within. While the lack of a specific data model enables the support of legacy applications, it significantly reduces the flexibility in dealing with the data, since there are only files in the system. Files alone do not have significant structure. In fact, as discussed in [84]:

> "But, file systems have no metadata beyond a hierarchical directory structure and file names. They encourage a do-it-yourself-data-model that will not benefit from the growing suite of data analysis tools. They encourage do-it-yourself-access-methods that will not do parallel, associative, temporal, or spatial search. They also lack a high-level query language. Lastly, most file systems can manage millions of files, but by the time a file system can deal with billions of files, it has become a database system."

In the same paper, the authors discuss the convergence between distributed file systems and other systems:

> "There is a convergence of file systems, database systems, and programming languages. Extensible database systems use object-oriented techniques from programming languages to allow you to define complex objects as native database types. Files (or extended files like HDF) then become part of the database and benefit from the parallel search and metadata management."

The next design principle follows directly from this discussion. Because files do not provide significant structure, the distributed data management system includes one other data unit, which is the dataset. A dataset, which is formally defined in Section 4.4.1.1, is loosely defined as a collection of files that are usually used together. This matches the observation that users rarely use a single file in isolation but almost always make use of groups of files, grouped statically by some shared semantic concepts. I now define the second design principle.

Design Principle: The distributed data management system uses datasets, which are (loosely) defined as collections of files, as the unit for all user operations.

Throughout this work the advantages of having native support for datasets will be discussed with emphasis on the useful side-effect of datasets being collections of *semantically related* files. Datasets present similarities to the hierarchical organisation of directories and files on a file system but there are differences that allow for a more flexible behaviour than what is available in file systems. This is discussed in Section 4.3.3.

4.3.2 Replication and Consistency Model

The requirements from users (Section 4.2.1) and managers (Section 4.2.2) include the transfer of sets of data between data centres. For instance, users typically wish to have a copy of some remotely available data at their home data centre where they can analyse it repeatedly. Managers want to establish automated data flows for some of the requests that have a more permanent nature, so that e.g. all data of a certain type is promptly transferred to a specific data centre as soon as it is produced at some other data centre.

In fact, such *replication* is a commonly employed technique for increased reliability and performance, as discussed in [165]. Using replication, reliability is improved because if one copy of the data is not working, it is possible to use others. Performance can also be improved because it is possible to distribute large numbers of users among several replicas, increasing the total number of (parallel) accesses to the data. In addition, it is possible to use geographically closer replicas (geo-locality), so that each data processing step will use data in its proximity. These are, implicitly, the requirements of the users and managers. In the context of a large, worldwide distributed application, geo-locality is important because the network latency can be very significant.

Nonetheless, replication is difficult to implement. According to [165], there are two main issues with replication: managing replicas and keeping them consistent. The first issue is deciding where, when and by whom replicas should be placed. This problem is traditionally divided into two sub-problems: where to place the servers that will host the replicated content, and how to decide which of those servers to use during a replication process.

In the context of this work, the first sub-problem clearly has an easy solution: the replicated content will be stored in each storage at a data centre. Deciding which storage and data centres to use for placing a replica is a more difficult problem. This will be studied throughout this work, but the majority of times it will be users and managers that manually make these *replica placement* decisions. Hence, the task of the system will be to fulfil these requests as quickly as possible.

The other issue with replication is keeping replicas consistent. If two copies of a file are not the same due to failed replication, a data processing step may output different results, which can be problematic particularly if users are not made aware of the difference. The solutions to this problem are based on *consistency models*. There are multiple consistency models: data-centric or client-centric. For an overview of these, refer to e.g. [165].

In this work, the consistency model has two separate dimensions for each of the two data units in the system, which are the files and the datasets. I assume that files cannot be updated. Therefore, replicated files do not have any consistency issue: each replica of a file is consistent as long as the replication was successful. Nonetheless, the datasets, which are dynamic collections of files, can be updated. Files may be added or removed from a dataset. For datasets, I adopted for a consistency model based on *eventual consistency* [170].

Enforcing that files cannot be updated is not a major restriction for the type of processing envisioned in a data-intensive application. In fact, it facilitates the implementation of the user data provenance requirement. That is, whenever a file needs to be updated, a new file can be created. Users that depend on the old file will still be able to use it without changes. If a file (and therefore all its replicas) contains erroneous contents that must be replaced, a new file can be created and the old replicas deleted. If users depend on the old 'bad' file, they will not be able to find it. This results in an error condition that is actually preferable by the managers, instead of having a user algorithm silently using outdated files.

Alternative models that allow files to be updated would be difficult to implement and would suffer from significant latency. For instance, using either two-phase commit [83] or even a more advanced alternative such as the Paxos consensus protocol [109], requires the exchange of multiple messages to commit every update operation. Considering that replicas are in separate data centres, the network latency is substantial. In addition, if failures occur, there would be the need to synchronise replicas in the background. This process would be difficult to implement, particularly in an environment characterised by the need to handle large amounts of data during normal operation, where there might not be network resources left to synchronise data in the background. A simpler alternative, employed for instance by GFS [80], is to allow append-only operations to files. Nonetheless, the latency issues remain present. Also, given the introduction of the dataset as the data unit, there is no particular advantage in appending files when datasets can be "appended" by adding additional files.

While I assume that files can be easily (or rather, inexpensively) recreated, the same does not apply for datasets. The process of producing a dataset can be time consuming and involve multiple users. Therefore, it is desirable to allow updates to the contents of a dataset so that a single bad file does not render the entire dataset unusable. The eventual consistency model defines that if a dataset changes (e.g. by replacing a 'bad' file with a corrected version), all replicas of the dataset should eventually reflect this update.

Data stores that are eventually consistent have the property that, in the absence of updates, all replicas converge towards identical copies of each other. Updates eventually propagate and conflicts (if multiple writes are allowed) are typically easy to solve. In addition, applications can tolerate a degree of inconsistency between replicas, but should be aware of such inconsistencies. A widely used example of an eventually consistent protocol is SMTP [98] for email submission: the delivery of sent email is 'guaranteed' to occur but not within any well-defined time window.

Clearly, an eventually consistent model is not applicable to all applications. Nonetheless, with the combination of immutable data files and the dataset as a data structure, this model is relatively easy to implement and provides sufficient functionality for most large scale, embarrassingly parallel applications.

Therefore, datasets are created by independent and geographically distributed processes. Because each dataset is an evolving collection of files, its constituent files will be gradually (eventually) replicated to all requested destinations, in a continuous manner. When the dataset becomes immutable (when it can no longer change), all replicas of the dataset will eventually converge to have the same set of constituent files. Section 4.4.1.1 elaborates on this issue. I finally introduce the relevant design principle.

Design Principle: The distributed data management system employs eventual consistency principles for replication and consistency.

4.3.3 Logical and Physical Data Units

The following design principle establishes a separation between logical representation of data and physical instantiation. That is, to further increase the flexibility for the users of the system and still optimise storage and network usage, there is a separation between the logical and the physical units of data handling. As a result, a dataset is a logical unit that is created independently of its constituent physical files.

This flexibility enables an easier integration with existing data flows, since datasets can be managed at a higher-level by separate processes, without altering the legacy applications that produce and consume the individual data files. At the same time, users can use datasets for querying and locating data or moving data across data centres.

This separation also allows the implementation of dataset replication based on eventual consistency. As the logical constituents of a dataset change, its (physical) replicas will eventually reflect this change. Future sections give examples of this behaviour, but I first introduce the corresponding design principle.

Design Principle: The distributed data management system establishes a separation between logical dataset definitions and physical replicas by using eventual consistency principles.

4.3.4 Data and Fabric Independence

An important principle from the data management theory is *data independence*. Data management systems, such as relational databases, provide two forms of data independence, termed *physical data independence* and *logical data independence*. Physical data independence is the ability to change the underlying physical data organisation without breaking any application programs that depend on it. In the case of a relational database, this could be re-partitioning rows in a table across multiple disks or nodes, transparently to the users. Logical data independence is the ability to insulate programs from changes to the logical design; if an underlying data schema is changed, views, which define virtual background compatible schema representations, can be created and used by existing programs.

Clearly, data independence is a desirable feature. In fact, it is reasonable to augment the definition of physical data independence to the entire distributed computing fabric (architecture of data centre, location of the data, etc), which results in the following design principle:

Design Principle: The distributed data management system employs *fabric independence*, which is the ability to change any part of the underlying distributed fabric transparently to the users.

From this definition, it is clear that managers and data centre administrators may be involved in these changes (e.g. managers must authorise new data centres to join the system, and data centre administrators are actually involved in changing the underlying fabric). Hence, the principle is only applicable to the users.

4.3.5 Layered System

The final design principle concerns the layering of the distributed data management system on top of existing storage middleware. This principle, which follows from the data centre administrator requirements, is not applicable in the realm of data management theory and is specific to a complex distributed environment.

Given the requirement by data centre administrators for minimum intrusiveness, it is desirable that the distributed data management system does not require any changes to the storage already deployed at a data centre. Instead, the distributed data management system must be layered on top of the existing storage, i.e. on top of an existing NAS/SAN or HSM, by defining an abstract layer for all storage interactions. In addition, this principle considerably extends the ability to make opportunistic use of storage resources. The disadvantage is that a layered integration can lead to many more potential inconsistencies and a more complex design. These issues are discussed throughout Chapter 5.

Design Principle: The distributed data management system is layered on top of existing storage middleware and provides a single and unified interface to their aggregate capabilities.

Following from this principle and similarly to a trend observed in recent distributed file systems (discussed in Chapter 2) there is no enforcement of POSIX semantics on dataset operations. Users of the system require specific tools to access and manipulate datasets. This allows for greater flexibility in the design of the system, particularly given the introduction of new data units and a layered approach.

4.4 Architecture

This section describes the architecture of the distributed data management system. I begin by describing formally the system entities, which are the datasets, logical and physical file names. This is followed by an overview of the system functionality, which includes a discussion on two major architectural decisions regarding dataset cataloguing and naming and subscriptions. The last section contains a description of the components in the system and an overview of their interactions.

4.4.1 System Entities

4.4.1.1 Datasets

A dataset has been loosely defined as a dynamic collection of files. A more precise definition is:

Dataset. A dataset is a collection of files (typically containing more than one physical file) that are processed together and usually comprise the input or output of a computation or data acquisition process.

A dataset is, at the lowest level, file metadata: a file is assigned as being part of one or more datasets. This attribution provides very useful properties: knowing that a dataset represents files that are used together, the system can optimise its units of data transfer and discovery. Locating datasets as opposed to files implies storing far fewer entries on a database, hence improving overall scalability.

Similarly, when transferring data, the dataset provides very good ordering of requests: if there is a long queue of data to transfer, it makes the most sense for users to have the system transfer those files that allow users to advance with their analysis as soon as possible. As such, the system must try to transfer the missing files from a dataset as soon as possible, to complete the replica of the dataset.

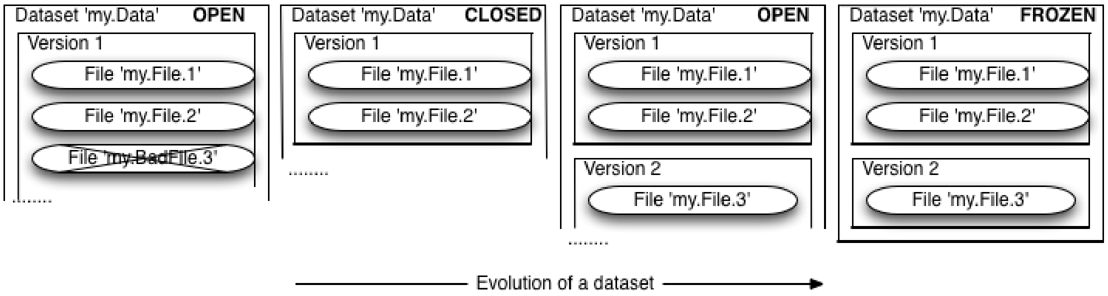


FIGURE 4.4: Evolution of a dataset.

Additionally, there is often the need to assign metadata attributes (e.g. software version used to produce the output) to a set of files. For scalability reasons, it makes the most sense to assign a single metadata attribute to a dataset as opposed to assigning it individually to a set of files. External metadata systems can then use datasets to reference data.

Creating a dataset is typically a highly parallel task, where jobs running in a computing cluster produce the constituent files. To facilitate the iterative process of constructing a dataset, which may last a long time[3], the architecture defines the possibility to create versions of a dataset along with dataset states.

A dataset can have multiple versions. Newer versions of a dataset can either add or remove files from the dataset. Dataset versions allow users to reference a static set of files at any point in time. For instance, earlier versions of a dataset can contain only a smaller subset of the complete data sample. Nonetheless, to have reproducible analysis, it is important to re-use the same exact sample. Versions can also be used to replace existing files by corrected or improved versions, if the original files contained errors.

Datasets have three states: open, closed or frozen. Whenever a dataset is created, the dataset state is *open*. At this point, files may be freely added or removed from the dataset. The dataset can then be *closed*. When it is closed, the dataset definition cannot change: files cannot be added or removed from the dataset but the dataset may be re-opened again by adding a new dataset version. Dataset versions can only be defined when the dataset state is closed. Finally, when the dataset production is completed, the dataset state is set to *frozen*. At this point, the dataset cannot be re-opened again, since no new versions can be defined: it is immutable[4].

Figure 4.4 illustrates the mechanisms of versions and states. A dataset is represented at four points in time, with the earliest representation on the left side and the latest on the right side. Initially, the dataset is shown on state `OPEN`. It has three files in its `Version`

[3] For instance, within ATLAS, it is common to have the production of a dataset last several weeks.

[4] The dataset definition cannot be changed if the dataset is frozen, but it is possible that some files are lost. As discussed in Chapter 5, temporary losses may be automatically recovered in some situations. Permanent losses may require the re-definition of a smaller dataset.

`1`, but one of the files (badly produced) is removed whilst the dataset is open. Because the version is open, files can be added and removed. At a later point, the dataset is `CLOSED` by the user. This is the second representation in the figure. At this point, users can reference `Version 1` of the dataset `my.Data` to reference those two files. But at a later point (third representation), a new dataset version is added. From now on, the latest version is `Version 2`. Users can continue to reference `Version 1` of the dataset to reference the two initial files, but only the latest `Version 2` can be changed. Finally, in the fourth representation, the dataset is `FROZEN` by the user. Now, `Version 2` cannot be changed any more and in addition, no new versions can be created. The user can reference either `Version 1` (two files) or the dataset itself (`Version 2` with three files).

To maintain the integrity of the dataset in a distributed system, there are no guarantees about the set of contents available at a specific storage whilst the dataset is not frozen. Only some time after the dataset is frozen (i.e. when no further changes are allowed), will the system be able to guarantee that copies of the dataset are the same. This follows the eventual consistency model described in Section 4.3.2. A correlation can be established with this model and the last-close-wins semantics of distributed file systems, applied to a higher-level concept.

4.4.1.2 Logical and Physical File Names

I now define logical and physical file names.

Logical File Name (LFN). A logical file name is a location independent human-readable name that uniquely and uniformly identifies a set of file replicas. File replicas are exact copies of a file. The initial copy is usually called the *master copy* but, with a slight abuse of terminology, all copies will be called replicas regardless of which one is the master.

Physical File Name (PFN). A physical file name is a location dependent access protocol and path information that can be used to physically access a data file. Unlike the LFN, the PFN may or may not be human-readable.

4.4.1.3 Example

This section illustrates the concept of dataset, logical and physical file names with an example from the ATLAS Experiment. During a run of the ATLAS Experiment, files are continuously written to the storage at CERN. Some of these files share many common properties, because they are part of the same *physics stream*. The name of the physics stream name, which is also the dataset name, is chosen to be `atlas.mc12.00001.AOD`.

During the run, two files for the physics stream are written to the storage at CERN. These are:

```
/atlas/data/atlas.mc12.00001.AOD.1
/atlas/data/atlas.mc12.00001.AOD.2
```

These are physical file names, because they are specific to the CERN files. These physical files do not include any information on the access protocol. In this case, POSIX access is assumed. Because these files are going to be transferred to other sites, their new access protocol[5] becomes:

```
gsiftp://srv01.cern.ch/atlas/data/atlas.mc12.00001.AOD.1
gsiftp://srv01.cern.ch/atlas/data/atlas.mc12.00001.AOD.2
```

Because it is desirable to refer to these files without referring to any particular replica, two LFNs are created. The LFNs for these two files are chosen to be:

```
atlas.mc12.00001.AOD.1
atlas.mc12.00001.AOD.2
```

At this point, these two logical file names are added to the (logical) dataset `atlas.mc12.00001.AOD`. Later, the dataset is transferred to a remote storage, for instance, to the Rutherford Appleton Laboratory (RAL). To refer to the copies at the RAL storage, two physical file names are used. For instance, these could be:

```
gsiftp://storage.ral.ac.uk/atlas/1234-4567-AAAA-1234
gsiftp://storage.ral.ac.uk/atlas/1234-4567-BBBB-5678
```

(Note that physical file names are not necessarily human-readable, unlike logical file names or dataset names, and the internal paths and names are usually different across storages.) Therefore, the dataset is a logical entity that contains logical file names. These are location-independent. If a file is available locally, then there is a unique physical file name for each available logical file (i.e. for each replica). Finally, if a user wishes to refer to any copy of the first file in the dataset, the user must use the logical file name `atlas.mc12.00001.AOD.1`. Otherwise, the user must choose one specific storage and use the corresponding physical file name.

4.4.2 System Functionality

This section contains a non-exhaustive list of user functionality. This functionality is based upon the design principles and system entities introduced in previous sections. (This functionality is described in more detail in later sections as well as in Chapter 5.) All functions take a dataset name as input parameter, following the design principles introduced in the previous sections.

[5] Details on access protocols such as `gsiftp` (GridFTP) are presented in Chapter 5.

- **Dataset creation.** Users can create datasets in the system. The dataset can be modified by adding or removing files, using specific tools to upload user files to the storage and to add the logical file names to the dataset. Users do not control or manage the physical location of the files within the storage. This is done internally by the system since the fabric independence principle states that the computing fabric should be used as a "black-box".
- **Dataset transfer.** Users can request the transfer of datasets between storages. This is one of the main functions of the system.
- **Dataset deletion.** Users can request the deletion of datasets from storages.
- **Dataset queries.** Users can query the datasets available in the system, its constituent files and other system attributes.
- **Dataset retrieval.** Users can retrieve an entire dataset or parts of a dataset to a local storage outside of the system's control (e.g. to the user laptop).
- **Dataset notifications.** Because datasets are typically very large (GBs or TBs), operations such as transfer or deletion can take a long time. The system can generate notifications when these operations begin and/or are completed. For instance, when the transfer of a dataset is complete, a notification can be sent to an external job submission system that automatically dispatches jobs to the computing cluster, so that the newly available files are processed.

This functionality encompasses both bookkeeping and data distribution, first illustrated in Figure 1.2 of Chapter 1. For the bookkeeping dimension, the functionality requires a catalogue that uses the dataset as its underlying data unit. This is the subject of the following section. For the data distribution functionality, the system requires a service that is tightly coupled with the cataloguing, implements the replication and consistency model previously described, enforces the separation between logical and physical data units, implements the fabric independence principle and is based on a layered architecture. This is the subject of the second section.

4.4.2.1 Cataloguing and Naming

The distributed data management system requires a catalogue that knows all the datasets in the system. This catalogue is contacted to create datasets, dataset versions and change dataset states. Whenever the system performs any operation on the dataset, it also reads the latest dataset definition from the catalogue.

A concern with the dataset catalogue is the large number of entries it is expected to hold - all datasets and all constituent files - and the large number of accesses. Services and users also require a highly available catalogue. Therefore, a solution based on a single central catalogue is not adequate, because this can result in a single point of failure. The alternative is to employ multiple instances. In this scenario, there are two main alternatives. One is to have replicated catalogue instances across multiple nodes, either using a primary-backup[6] scheme [7], where a node is declared to be the master and all other nodes act as backups, or a state machine approach [154], where a protocol ensures that all nodes eventually reach the same state and hold the same information. The second alternative is to use some form of *partitioning*, so that different nodes contain different subsets of the catalogue.

I have chosen to employ partitioning given its simplicity as compared to the implementation of primary-backup schemes or state machine consensus on the wide-area network. Both alternatives are more suitable within a data centre environment with low network latency. This is similar to the discussion on Section 4.3.2, regarding the replication and consistency model, and where systems such as Paxos are said to suffer from significant latency on the wide-area network given the required message exchanges. Having decided to use partitioning, I describe next what is the partitioning criteria, by discussing dataset naming.

In a distributed system, it is the responsibility of a naming service to resolve a name to its location. It is of interest to design a naming service that explores the distributed environment and allows for partitioning the catalogue, avoiding single points of failure and increasing scalability. Following the terminology introduced in [165], there are three alternative designs for a naming service: flat naming, structured naming and attribute-based naming. I briefly discuss each of these alternatives before motivating the choice of structured naming.

Flat naming consists of having unstructured names. It can be implemented using simple solutions such as broadcasting, where a query is forwarded to every (catalogue) instance in the system, and the instance holding the entry replies back. This results in additional communication overhead for queries.

Another alternative for flat naming is the home-based approach. This consist of defining a static home location for every entry (in this case, for every dataset name), which is typically where it was first created. If an entry is moved to a different (catalogue) instance, the original instance will keep track of the current location. This solution can result in imbalances in the number of stored entries per instance and additional communication overhead.

A more robust alternative for a flat naming service is based on distributed hash tables or DHTs. DHTs were introduced in Chapter 2 and can be used to resolve a name to an associated entity. These have been successfully applied to large peer-to-peer networks as discussed in Section 2.2.1.

[6] Also known as master-slave scheme.

Structured naming services are based on the definition of a well-defined structure for each name in the system. Unlike flat naming schemes, which may not be human-readable names, structured naming is specifically targeted to be readable by humans. Structured naming requires the definition of a name space. A name space can be represented as a labeled, directed graph with two types of nodes: the leaf nodes, which correspond to the named entities, and the directory nodes. The name resolution consists of navigating through the graph, from its root to a named entity, following the directory nodes.

Structured naming is used in almost all distributed file systems, and results in a simple hierarchical scheme, although extensions such as symbolic links are often used. An advantage of structured naming, besides being human-readable, is the possibility to distribute the name space, similarly to DHTs but with a considerably simpler implementation.

Attributed-based naming is a different technique from the previous two, in that each entity is associated with a set of attributes. Each of the attributes partially describes the entity. It is mostly useful for containing metadata information about the named entities, but queries are considerably more complex than those described in the previous solutions. Attribute-based naming is most appropriate for a higher-level metadata service, that builds on top of the dataset catalogue, where users can locate datasets based on its attributes.

Based on the previous discussion, the most appropriate choices for the naming service are either DHTs or structured naming. I have opted for structured (dataset) naming for two reasons. The first reason is that structured naming enables catalogues to be partitioned into independent instances. Each instance stores a portion of the name space. Structured names make such partitioning easy to implement. The second reason is that the added complexity of DHTs is not required. DHTs are useful for dynamic environments, while it is assumed that catalogue instances in this system are stable. If some new instance needs to be deployed, this can be coordinated by the application managers. With this choice I assume that it is simple to define a partitioning of the name space. In practice, managers are able to define stable structures for the dataset name, and assign parts of the hierarchy to different catalogue instances.

As an example, in the ATLAS Experiment, the name `mc.12.AOD.00001.root` is assigned to the catalogue instance for the Monte Carlo data, or `mc`. Because these data correspond to ATLAS-wide Monte Carlo, only authorised users are able to write under the `mc` area, so that no users can mistakenly use (or abuse) this name. Note that unlike in traditional file systems, the separator employed in this example is '.' and not '/'.

4.4.2.2 Subscriptions

As discussed in Section 4.4.2, transferring datasets is one of the main functions of the system. Nonetheless, deletion and other dataset operations are equally important in a large distributed environment. This section describes the motivation for using the concept of *subscriptions* for managing physical instances of datasets, similarly to the principles introduced in [150] and discussed in Section 2.3.1.

A subscription is a persistent request, issued by a user, to a storage and for a dataset. If a storage is subscribed to a dataset, the system will ensure that all files in the dataset will get transferred to the storage. If the dataset changes (i.e. while the dataset is not frozen), with the addition and removal of files, the subscription will ensure that the corresponding changes are eventually applied to the local copy at the subscribed storage. That is, if a file is added to a dataset, all subscribed storages will eventually copy the file over. If a file is removed from the dataset, all subscribed storages will remove the file, if it is not part of any other subscribed dataset.

As such, subscriptions are an adequate mechanism to implement an eventual consistency model. Subscriptions also follow the split between logical and physical data units: the contents of the dataset can change, at the logical-level, without any synchronous interactions with physical replicas. Synchronous interactions would likely be affected by high latency on the wide area network.

Also, subscriptions are used to implement complex data flows. Because the process of creating a dataset can take a long time, managers can subscribe datasets immediately after their creation. As files are produced, uploaded to a storage and added to a dataset, these will automatically get copied to all subscribed storages. This mechanism simplifies the process of managing many transfers with pre-defined data flows, which is a managerial requirement.

In addition, subscriptions improve the reliability of the system, as they allow the transfer of data to be an asynchronous process. Subscriptions are executed by processes that are running in the background, as presented in later sections. Asynchronous transfers augment the perceived reliability of the system, because a temporary failure can be retried without user intervention.

Subscriptions also allow for a better scheduling of transfers, which is desirable for a data-intensive environment. The transfers for a storage can be re-scheduled and delayed internally, avoiding overload conditions. These features would be difficult to implement if users were synchronously transferring data.

Finally, the persistent nature of the subscription concept provides useful properties. For instance, if there is an occurrence of lost data at a storage (e.g. due to a hardware failure), the system will eventually detect the missing files and copy them over automatically,

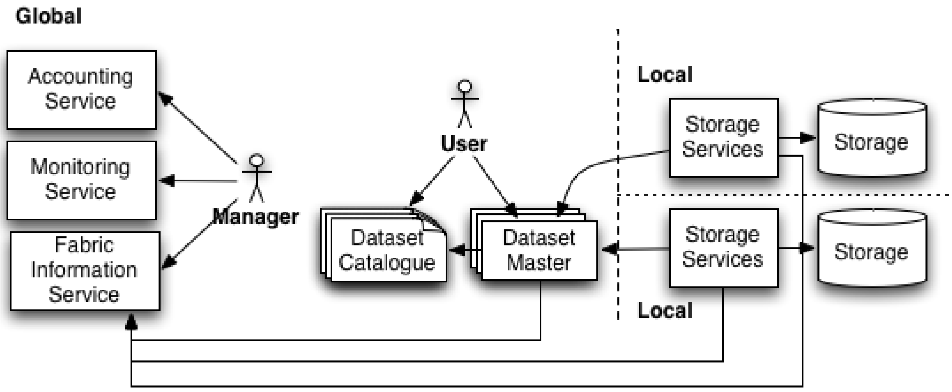


Figure 4.5: Overview of the system architecture.

as long as the storage is still subscribed to the dataset. That is, while a subscription exists for a dataset and storage, the storage is supposed to hold a complete replica of the dataset or be in the process of transferring missing files.

The next section describes the components in the architecture, and presents a high-level overview of the implementation of dataset catalogues and subscriptions.

4.4.3 System Architecture

Figure 4.5 illustrates the architecture of the distributed data management system. As shown in the figure, the architecture uses a combination of global and local services. Global services are responsible for higher-level functionality, such as the creation of datasets. Global services store and operate mostly[7] with location-independent information (i.e. dataset names, logical file names), while local services store and operate with location-dependent information (i.e. physical file names). Local services are also responsible for executing lower-level actions on files, such as the deletion of files from a storage.

As shown in the figure, users interact with a component called the *dataset master*, which is internally partitioned into multiple instances. The dataset master is the global service that implements the user interface described in Section 4.4.2. The dataset master is also used by local services and other global services.

Another global service is the *dataset catalogue*, which is responsible for storing the definition of all datasets in the system. This is introduced in Section 4.4.3.1 and is internally partitioned as discussed in previous sections. The figure shows additional global services: the *fabric information service*, the *monitoring service* and the *accounting service*. These are introduced in Section 4.4.3.4, and are primarily *administrative services*

[7] Global services do hold some local information, but only as a caching mechanism. This is discussed in later sections and in Chapter 5.

used by managers or by other components in the architecture.

Local services are responsible for interacting with the storage at each data centre to perform the transfer, deletion and lookup of files. Therefore, local services are also called *storage services*. There is an instance of a storage service per storage system; if a data centre contains two separate storage systems, it runs two independent storage services.

All operations performed by the storage services require local information. For instance, to physically delete a file from a storage, the storage service needs to have native support for the storage deletion commands, determine the disk servers hosting the file and its absolute path. In the next sections and in Chapter 5, the internal architecture of the storage services is described in detail.

The split between global and local services allows for a more robust design, employing the classical principle of *separation of concerns*, as it enforces local information to be kept only locally. This facilitates changes to the system - e.g. if a host name changes, only a local service is affected; and minimises security exposure - e.g. host names are not known outside the local data centre environment. These principles follow the data centre administrator requirements and the principle of fabric independence.

Similarly, the split between global and local service enables a separation between the logical and physical units of data. Logical dataset definitions are kept globally, while the exact location of its constituent files are only known locally. As such, the files part of a dataset can change at the logical level without any (synchronous) interaction with local services.

The next sections describe each of these components in more detail. (Additional design and implementation details are also given in Chapter 5.)

4.4.3.1 Dataset Catalogue

Each instance of the dataset catalogue stores the definition of some of the datasets in the system, according to the partitioning and structured naming described in Section 4.4.2.1. The dataset definition includes the dataset name, which is a human-readable string, the dataset state (`OPEN`, `CLOSED` or `FROZEN`), an integer with the latest dataset version number, and the set of logical file names in each version. An example of a dataset definition is given in Listing 4.1.

The signs `+` and `-` represent the files added and removed in each version, as compared to the previous version. Because it is expected that dataset versions represent an evolution of the dataset, only the difference in files between consecutive versions are stored, as opposed to the full list of files per version.

```
Dataset name: mc.12.AOD.00001
State: OPEN
Latest Version: 3
Changes in Version 1
 +  mc.12.AOD.00001._01.pool.root
 +  mc.12.AOD.00001._02.pool.root
 +  mc.12.AOD.00001._03.pool.root
Changes in Version 2:
 -  mc.12.AOD.00001._03.pool.root
 +  mc.12.AOD.00001._03.v2.pool.root
 +  mc.12.AOD.00001._04.pool.root
 +  mc.12.AOD.00001._05.pool.root
Changes in Version 3:
 -  mc.12.AOD.00001._04.pool.root
 +  mc.12.AOD.00001._04.v2.pool.root
 +  mc.12.AOD.00001._06.pool.root
```

Listing 4.1: Example of a dataset definition.

To know the contents of version 3 of the dataset, the system needs to read all dataset versions in order, and apply the additions and removals of files (given by the + and - signs). This scheme results in reduced storage space and is adequate for an environment where file additions are more common than replacements. For instance, for Listing 4.1, version 3 of the dataset contains the files:

```
mc.12.AOD.00001._01.pool.root
mc.12.AOD.00001._02.pool.root
mc.12.AOD.00001._03.v2.pool.root
mc.12.AOD.00001._04.v2.pool.root
mc.12.AOD.00001._05.pool.root
mc.12.AOD.00001._06.pool.root
```

There are additional system attributes stored for each dataset definition and logical file name, such as the dataset owner or the file size. These are detailed in Chapter 5. Nonetheless, note that the dataset catalogue has no information on physical file names; instead, it stores only logical, location-independent information.

4.4.3.2 Storage Services

Storage services are the background processes, deployed per storage, which are responsible for the management of the (physical) files in a storage. Managing files, or replicas, includes the following tasks:

- **Lookup.** The storage services include a component responsible for looking up files in the local storage. This requires translating a logical file name to a physical file name, and checking in the storage whether the files are readily available.

- **Transfer.** The storage services include a component responsible for transferring files from a remote storage to the local storage. Note that each storage service is assigned to a single storage, and the destination storage is the one triggering the transfer request. A transfer can only occur between different storage services, which may be located in different data centres on the wide-area network or even within two separate storages within a data centre.
- **Deletion.** The storage services also include a component that deletes files from the local storage.

Each of these tasks is performed by separate services within a storage service. All tasks require a tight coupling with the storage, which includes interfaces to lookup, write and delete files. The coordination of which files to lookup, transfer or delete is handled by the dataset master described in the next section. Storage services continuously poll the dataset master for additional work. The reply from the dataset master is a work assignment, which includes a list of logical file names on which to perform the lookup/delete/transfer task.

For example, the delete component of the storage service asks the dataset master for files to delete locally. The dataset master returns a list of logical file names (LFNs). The delete service resolves each LFN to a physical file name (as shown in Chapter 5) and then physically deletes the file from the storage. When the deletion is completed, either successfully or failed, the result is sent back to the dataset master.

This work assignment model respects the separation between logical and physical data units, since the physical file names and internal storage configuration settings are only used locally. It also ensures that storage services are not aware of datasets definitions, which can change globally.

In addition, the work assignment model allows administrators to locally override any decisions. For instance, the administrators can disable the deletion component or prevent it from deleting specific files, in effect applying local policies. It also allows administrators to locally throttle the number of accesses to the storage. For instance, when transferring files from a remote storage, the transfer component may have received a list of 20 files to transfer, but decide to transfer the files in 4 blocks of 5 parallel transfers, hence limiting the number of local parallel storage accesses to 5.

Chapter 5 discusses the request handling mechanism in greater detail, along with the built-in authentication, authorisation and fault tolerance mechanisms.

4.4.3.3 Dataset Master

This section describes the dataset master, which is the architectural component that implements the user interface described in Section 4.4.2. As shown earlier in Figure 4.5, the dataset master interacts both with the dataset catalogue and the storage services. The figure also illustrates a deployment scenario where there is more than a single dataset master. The architecture foresees the deployment of multiple independent instances of the dataset master, where each instance acts as the exclusive 'master' for a subset of the datasets in the system.

When a user requests a dataset operation (e.g. the subscription of a dataset to a storage), the request is sent to a unique dataset master instance. This redirection uses a service that guarantees unique mapping between a dataset and a master service, which is described in Chapter 5. The dataset master will then handle the request.

As discussed in the previous section, the dataset master does not execute any of the user requests directly. Instead, it assigns work to the local storage services. The local storage services are not dataset-aware, but only work with bulk requests of files to lookup, transfer or delete. It is up to the dataset master to define these work assignments.

For example, suppose a user requests the subscription of a dataset to a storage. The request is sent by the user and queued in the dataset master. In the background, the transfer storage services are polling the dataset master for work. The dataset master will check the queued requests, resolve the files in the dataset by contacting the dataset catalogue, and reply with the list of files to transfer. In practice, the exact mechanism is slightly more complex, as it involves asking other storages (the sources) whether they have replicas of the missing files. These workflows are described in Chapter 5.

Although the dataset master is a global component, the architecture achieves a good scalability as shown in later chapters. In particular, the dataset master is partitioned into several independent instances, and each instance stores entries related to the actions currently active. If instead a global component were to receive individual file requests, the request rate would be considerably higher and could cause scalability problems.

Nonetheless, the dataset-based interface does not prevent the dataset master knowing about logical files and even physical file names. In fact, the dataset master stores such information but only in a transient manner. For instance, when the dataset master assigns lookup tasks to storage services, it gradually builds a cache of replica information from all the responses it receives. All this information is transient: if the dataset master restarts, it can simply reassign new lookup tasks to find the files.

This mechanism has the advantage of avoiding the need of a separate catalogue to keep the location of the files in the system. Such a catalogue could lead to scalability and consistency problems because any stored information could be incorrect. In a distributed system, a location catalogue is never absolutely correct as data can be lost unexpectedly. The described mechanism minimises these occurrences with minimal extra effort, by using responses from the storage services to build up a cache.

Chapter 5 elaborates on the interactions between dataset master, storage services and dataset catalogue.

4.4.3.4 Administrative Services

This section describes the remaining global services in the architecture. These are the following administrative services:

- **Fabric Information Service.** This service aggregates high-level information about the distributed computing fabric, such as the human-friendly designation of a data centre (e.g. ‘CERN’). Since this is a global service, administrators can globally add or remove (permanently or temporarily) data centres from the distributed computing fabric. This is also where the users and groups of the system are registered, along with additional information such as user quotas.
- **Monitoring Service.** The monitoring service is responsible for providing real-time and dynamic information on the system usage, based on information from the dataset master and storage services.
- **Accounting Service.** The accounting service provides historical information on the system usage. It uses the monitoring service and the dataset master as its information sources.

Further details on these services are presented in Chapter 5.

4.5 Discussion

This chapter proposes a novel distributed data management system, starting from design principles and defining the system architecture. Before describing the system in detail in the next chapter, it is appropriate to discuss the general applicability of this proposal.

The proposed system is designed primarily for data-intensive applications. Nonetheless, nothing precludes its usage in an environment with much less data (e.g. terabytes or gigabytes of storage in total, as opposed to petabytes). In fact, within the ATLAS environment, the same data management system that is used to manage the petabytes of ATLAS data is also used to manage log files, distribute binaries of the ATLAS software and even to make automatic off-site backups of relational databases. All these different data are added into datasets and managed as any other data.

Nonetheless, there are some important constraints to consider in its applicability. One is the number of data centres being served and the size of each data centre. If the system were to be deployed in individual user desktops (where each desktop would host a “storage service”), the system would likely suffer from performance issues unless a large number of dataset masters would be deployed. Nonetheless, the presence of a very large number of dataset masters would mimic some existing P2P architectures, hence indicating that such scenarios could in principle be successfully deployed but not without considerable implementation changes.

Another important constraint is the consistency requirements for updating and reading the data. In data-intensive applications, I have not encountered the need for transactional support when handling very large sets of data. Complex applications such as ATLAS have this need, but only for very small subsets of data (megabytes) in very specific scenarios, such as the systems that control and monitor the environmental conditions of the ATLAS detector during real-time operations. There is no considerable difficulty in these scenarios because they are not data-intensive nor require worldwide distribution of data.

Nonetheless, it is theoretically possible to envision a large scale data-intensive application that requires transactional support for all its petabytes of data stored around the world. In this case, the proposed system cannot readily be used. In practice, I believe those applications can be adapted to remove or relax the transactional support. I believe any alternatives with true transactional support would likely suffer from latency limitations, at the very least from the limits of the speed of light. In this proposal, the availability of datasets and the support for immutable files should, in principle, suffice for most scenarios. These concepts should allow even the applications with the most stringent requirements to be supported.

4.6 Summary

In this chapter, I described the design and architecture of a distributed data management system for data-intensive applications. This started by listing the requirements from users, managers and data centre administrators. These requirements include the need to move large sets of data between data centres, ensure that the operational overload of the system is small and that the system is minimally intrusive.

The design principles were also introduced. These include the introduction of datasets as a native data unit, a separation between logical dataset definitions and physical replicas, which implement an eventual consistency model, the independence from fabric changes and the layering of the system over existing storage middleware. Alternative techniques for dataset cataloguing were discussed and the choice of partitioning and structured naming was motivated, given its scalability and simplicity of implementation. Similarly, subscriptions were introduced as the mechanism to request movement of datasets between storages.

In addition, I presented the system components, which include a mix of global and local components. Global components are the dataset catalogue, the dataset master and a set of administrative services for fabric information, monitoring and accounting. Local services are also called storage services, and are deployed per storage system. Global services operate primarily with logical dataset definitions and local services with physical file replicas. All these components are described in detail in Chapter 5.

Finally, I discussed the general applicability of this proposal. I argued that this proposal can in principle support a wide-range of scenarios even if primarily oriented for large scale data-intensive applications.

Chapter 5

System Design

"Thou hast seen nothing yet."
(Miguel de Cervantes in *El ingenioso hidalgo don Quijote de la Mancha*)

This chapter describes the design and implementation details of the distributed data management system presented in Chapter 4. After introducing the methodology in the next section, I describe the dataset catalogue in the second section, followed by the storage services in the third section. The fourth section describes the dataset master, and focuses on the interactions with both dataset catalogue and storage services. The last sections briefly describe the various administrative services and the client tools. The fault tolerance and scalability properties as well as security considerations are discussed and reviewed in dedicated sections, given their importance to the design of the system.

5.1 Methodology

The system described in this chapter is the result of several design iterations. The first prototype was produced in 2005. Since then, and following the results of several large scale tests, parts of the system were redesigned. (One of these large scale tests is discussed in the next chapter.) Design changes were applied when limitations were identified during the large scale tests. For instance, early versions of the dataset catalogue consisted of a single centralised database. When this was perceived as a performance bottleneck by the database administrators, the naming service and vertical partitioning were introduced. Due to the large uncertainty in predicting future system load, design revisions were always preferred to hardware improvements, given performance results obtained during real operational conditions. While a new database server can significantly improve a catalogue's performance, application-level partitioning clearly provides another level of flexibility and ability to accommodate additional load. Nonetheless,

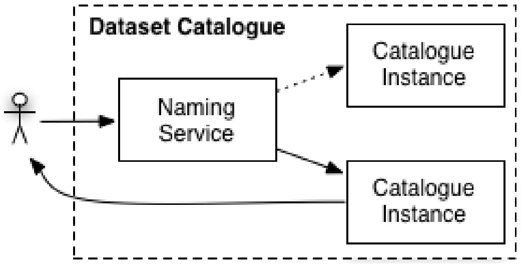


FIGURE 5.1: Overview of the dataset catalogue.

throughout the past years, the design principles introduced in Chapter 4 have remained unchanged and underlie all implementation decisions.

The design described in this chapter has been implemented by a group of developers who are part of the ATLAS Distributed Data Management project, which I led from June 2005 to March 2009. These include Pedro Salgado (Dataset Catalogue), Vincent Garonne (Dataset Master), Mario Lassnig (Client Tools), Ricardo Rocha (Monitoring Service), Fernando Barreiro (Accounting Service) and myself (Storage Services, Fabric Information Service, plus contributions to the Dataset Catalogue and Dataset Master).

The software development started in June 2005 and the first prototype versions were available by October 2005. It has since been used by the ATLAS Experiment to manage all its experimental data. The code is developed in the Python[1] programming language.

5.2 Dataset Catalogue

The dataset catalogue is responsible for storing the definition of all datasets in the system. Figure 5.1 illustrates the design of the dataset catalogue. Following the discussion in Section 4.4.2.1, it is divided into two components: a centralised naming service and a set of independent catalogue instances. The centralised naming service maps the dataset name to a unique catalogue instance that holds the corresponding dataset definition. Both the centralised naming service and each catalogue instance implement the same interface. Therefore, a request sent to the centralised naming service is transparently redirected to the appropriate catalogue instance.

The next section describes the relational database schema for the catalogue instance, and discusses the functionality available. The following section describes the primary and secondary dataset catalogue interfaces. The third section introduces the centralised naming service, which redirects user requests to the appropriate catalogue instance. The final sections discuss security mechanisms and review important properties of the design.

[1] Refer to *http://www.python.org*.

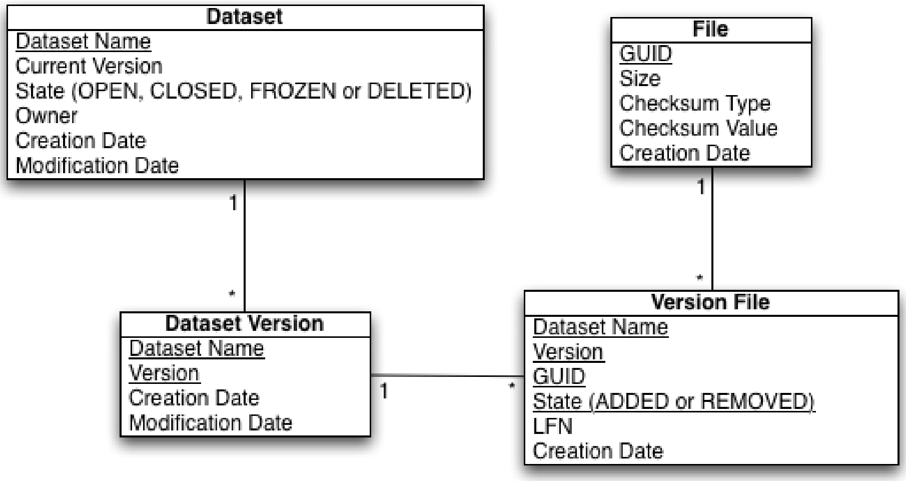


Figure 5.2: Schema of the dataset catalogue (primary keys are underlined).

5.2.1 Schema

The dataset catalogue relational database schema is shown in Figure 5.2. The `Dataset` table contains the dataset definition, the `Dataset Version` table contains the versions for each dataset and the `Version File` table contains the files in each version. Because a file may be part of multiple datasets or versions, the `File` table contains file attributes, such as file size and checksum. Both these attributes are per file regardless of the datasets or versions that contain the file.

The following example illustrates the schema usage. When a user creates a new dataset, an entry is created in `Dataset` and the state is set to `OPEN`. The first dataset version is also created in `Dataset Version`. At this point, the dataset does not contain any files. Whenever a user adds files to a dataset, the interface (described in the next section) requires the user to specify the dataset name and the list of files, including the LFN, GUID, file size and checksum. The corresponding entries are added to `Version File` and (if they do not yet exist) to `File`. If the `File` entry already exists, the file size and checksum attributes are compared to the stored attributes and the request is only accepted if the attributes match. The interface also allows users not to pass the file size and checksum in the request, since this is not required for files already known to the catalogue. In this case, the `File` entry must already exist or the request is denied, since the dataset catalogue requires that every (new) file definition includes the file size and checksum.

Given this schema, the user may either use the GUID or the combination dataset name and LFN to uniquely reference a file in the system. This follows from LFNs being unique only within the context of a dataset (i.e. two different files may have the same LFN in different datasets). On the contrary, GUIDs are globally unique identifiers and can be always used to uniquely reference a file, regardless of the datasets the file is in. Finally, a file may be part of multiple datasets: the GUID is the same for every dataset that contains the file, while the LFNs may be different.

For instance, when a new file is produced, it is assigned a GUID, such as: `2fede3d3-d16c-4481-ba86-b8f1f46adb8d`. At some later point, this file is added to a dataset owned by Alice, called `alice.dataset`. Alice chooses a LFN for the file, such as `my.file`. Bob also wants the file in his dataset, called `bob.dataset`, but uses a different LFN such as `alice.file`. In this case, either the GUID can uniquely reference the file, or the combination `bob.dataset` and `alice.file` or `alice.dataset` and `my.file`. (In this example, it is assumed that the file is part of the latest dataset version, otherwise the dataset version number is also required.)

Each entry of `Version File` includes the `State` field with states `ADDED` or `REMOVED`. The states represent whether the file is added and removed in this dataset version, as compared to the previous version. Because it is expected that dataset versions represent an evolution of the dataset, only the difference in files between consecutive versions is stored, as opposed to the full list of files for every version. For instance, to know the contents of version 3 of a dataset, the system needs to read all dataset versions in order (versions 1, 2 and 3), and apply the additions and removals of files given by their states. While this schema requires additional processing at the application layer, it results in reduced storage space requirements and is adequate for an environment where file additions are more common than replacements, and where dataset contents evolve gradually between versions.

Also, whenever files are added to a dataset, it is necessary to check whether the GUID is already part of the dataset, so that the same file is not added (or removed) twice from the same dataset. This constraint is not shown in the schema but is verified by the application-layer.

In addition to the dataset states `OPEN`, `CLOSED` and `FROZEN` discussed in Section 4.4.1.1, the `Dataset` table also includes an additional state `DELETED`. This state is set when the dataset is deleted by a user. The objective is to prevent dataset names from being re-used, even after the dataset has been deleted. This follows from a data provenance requirement, to guarantee that results are reproducible. Otherwise, users could delete and recreate datasets with different contents but using an existing name. Because maintaining old deleted dataset entries can become a scalability problem, this state is only used depending on a configuration attribute set per dataset catalogue instance. As such, managers can decide which catalogue instances (i.e. which portions of the namespace) must implement this functionality.

The relational database schema is deployed in an ORACLE database[2]. The `File` and `Version File` tables are expected to contain the largest number of rows. These are

[2] Refer to *http://www.oracle.com.*

ORACLE Index-Organised Tables[3] (IOT). An IOT is a table whose data is stored in B-Trees (see e.g. [99] or [50]) index leaves, in sorted order, based on the primary-key of the table. As a result, changes to that data such as adding, updating, or deleting rows require an update to the index only, making them very fast for primary-key based queries.

5.2.2 Interfaces

The dataset catalogue implements two separate interfaces, a primary interface and a secondary interface. The primary interface is implemented using Remote Procedure Calls (for RPC, see e.g. [165]). It uses a custom protocol built atop of the HTTP protocol and is used for creating, updating and reading dataset definitions. The secondary interface is used exclusively for reading dataset definitions and is more suitable for integration with external systems. I start by describing the primary RPC-based interface. Only a subset of the methods are discussed; the complete listing is available in Appendix A.

The primary interface includes the methods required to create and manipulate dataset definitions. For instance, the methods to add and delete files from a dataset have the signatures `add_files(dsn, list_of_files)` and `delete_files(dsn, list_of_files)`, where `dsn` is the dataset name and `list_of_files` are the list of LFNs or GUIDs, as discussed in the previous section.

The interface is implemented using the `Apache`[4] HTTP server, `mod_python`[5] and GridSite[6]. This software stack, with the exception of GridSite, is commonly used for large scale systems, as it allows for easy development of client/server applications, with the robustness of a widely used server such as `Apache` and the flexibility provided by the HTTP protocol. The addition of GridSite (using the `mod_gridsite` Apache module) enables the usage of the Grid Security Infrastructure (GSI). The GSI usage is presented and discussed in Section 5.2.4.

The implementation uses a custom RPC protocol. The protocol is based on XML-RPC [111] but its syntax is simplified by removing the XML-based encoding. This decision allows the client/server communication to carry reduced payload by removing spurious information. This is particularly important for methods that carry large listings of files, such as `list_datasets` and `add_files` methods (refer to Appendix A for additional details). For instance, it is possible for a dataset to contain $O(100,000)$ files. This decision also allows using HTTP streaming in the server replies, as discussed next.

The usage of HTTP is also justified by several features it provides. Two important examples are the ability to use Web Proxy servers and the ability to stream replies. A

[3] Refer to *http://www.dba-oracle.com/t_index_organized_tables.htm.*

[4] Refer to *http://www.apache.org.*

[5] Refer to *http://www.modpython.org.*

[6] Refer to *http://www.gridsite.org.*

web proxy server, such as Squid[7], is a server that is located between the client and the application server. It implements filtering rules and the ability to cache replies from the application server. When a client contacts the web proxy server, using the same interface as for the application server, the web proxy server will either reply immediately with a (valid) cached reply obtained from a previous request or, if no reply is cached, will contact the application server, cache back the reply and send it also to the client. This cached reply is then available for future requests. The application server, using specific HTTP headers, notifies the web proxy server of the time validity of its replies.

In the implementation, web proxy servers are used to cache dataset search requests (such as `list_datasets`) and dataset contents (such as `list_files`), also described in Appendix A. In particular, whenever datasets become frozen, the contents no longer change. Listing files in a dataset will always return the same content. In this case, the web proxy server can indefinitely (or, for long periods of time) cache the reply, saving the application server from answering some requests. This provides added scalability, by exploiting functionality widely available in HTTP.

Similarly, the HTTP protocol provides the ability to stream replies. If a user asks for a long list of datasets or files in a dataset, the server can send portions of the reply directly to the client. This saves the server from having to load the entire reply from the backend relational database into the server memory before dispatching it to the client. This is a desirable feature because it reduces the memory consumption of the server, hence increasing its overall stability and ability to serve a higher number of simultaneous requests.

An alternative protocol that was considered was SOAP (Simple Object Access Protocol) [26]. SOAP is a mechanism to exchange structured information, which evolved from XML-RPC. Nonetheless the added functionality provided by SOAP was not considered relevant for this scenario.

Instead, building upon the underlying usage of HTTP, the dataset catalogue includes a secondary interface. This secondary interface uses the REST architectural style [69]. It is used for reading information (e.g. for listing datasets and constituent files) and not for writing (e.g. for dataset creation).

The REST architectural principle is based on the "separation of concerns" principle, with the goal of simplifying the implementation of clients and servers. REST provides a minimal connector semantic: the messages in REST must be self-descriptive, leading to stateless requests. One of the scalability gains achieved by this principle is the ability to cache information.

A central concept in REST is that of a resource, which is the source of information. Resources are identified by global identifiers or URIs (Uniform Resource Identifier) [22].

[7] Refer to *http://www.squid-cache.org.*

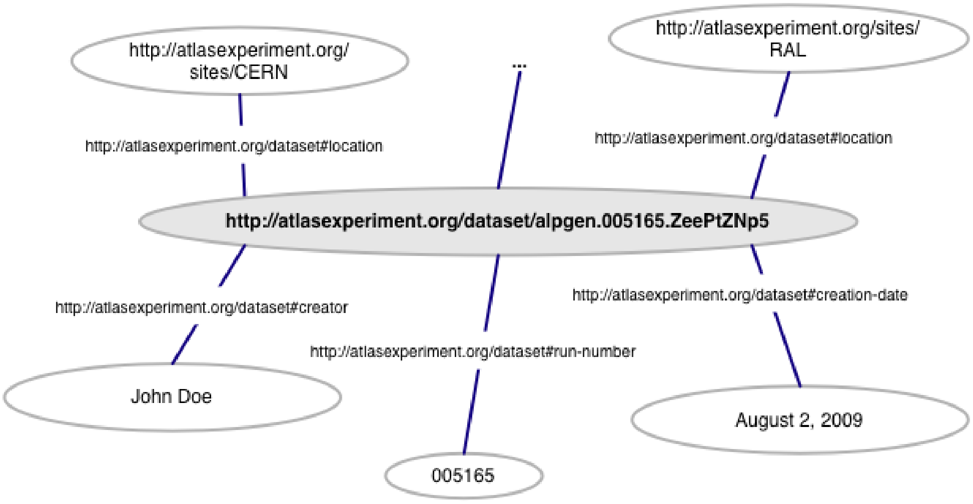

http://atlasexperiment.org/sites/CERN
...
http://atlasexperiment.org/sites/RAL
http://atlasexperiment.org/dataset#location
http://atlasexperiment.org/dataset#location
http://atlasexperiment.org/dataset/alpgen.005165.ZeePtZNp5
http://atlasexperiment.org/dataset#creator
http://atlasexperiment.org/dataset#creation-date
http://atlasexperiment.org/dataset#run-number
John Doe
August 2, 2009
005165

Figure 5.3: Example of RDF usage.

The World Wide Web uses a type of URI, which is the URL (Uniform Resource Locator) that specify where a resource is available and the mechanism to retrieve its representation.

The secondary dataset catalogue interface is based on assigning a URI to every dataset in the system. This allows dataset representations to be retrieved by URLs using HTTP. A dataset representation includes all the information on a dataset, such as its state, attributes, versions, constituent files and file attributes.

The availability of dataset identifiers allows external systems, particularly metadata systems, to be linked to the distributed data management system. Metadata cataloguing is not an integral part of the distributed data management system but users can exploit the available dataset URLs using additional mechanisms such as RDF (Resource Description Framework [126]) to build metadata data models.

An example of using RDF is shown on Figure 5.3, with a simple data model. Here, a resource that represents a dataset is assigned several properties. In RDF terminology these are resources, predicates and objects. By exposing the information about resources in the system using URIs, the implementation encourages external applications to link to the system, expanding its scope as a distributed data management system, layered on top of the Web architecture. In particular, as discussed in [28], this design enables an external metadata repository to store the provenance of the data known to the system.

5.2.3 Naming Service

The previous sections described the schema and interface implemented by each dataset catalogue instance. In this section, I describe the naming service, which allows for the existence of multiple dataset catalogue instances.

```
mc   http://mc-cat.cern.ch/
user http://user-cat.cern.ch/
```

Listing 5.1: Example of redirection rules.

The naming service is responsible for redirecting a request to the appropriate dataset catalogue instance. For scalability purposes, it is important to have more than a single dataset catalogue instance, because the schema shown in Section 5.2.1 requires uniqueness constraints and is expected to hold a large number of entries.

Every dataset catalogue instance implements the interface defined in the previous section. All methods include the dataset name in their signature. Dataset names, which follow a well-defined structure as discussed in Section 4.4.2.1, are used to redirect the request to a specific instance. The redirection of requests uses features from the HTTP protocol [70], more specifically the `HTTP status codes 3xx` that directs clients to go to another location.

An example of the set of static rules is given in Listing 5.1, where all dataset names starting with `mc` are redirected to a separate catalogue instance from those starting with `user`. The dataset naming structure, the number of catalogue instances and the redirection rules are statically defined by the application managers.

Besides its simplicity, an advantage of this schema concerns security issues and visibility of datasets. When the naming service receives a write request (e.g. `create_dataset` method defined in Appendix A), it can check whether the requester is authorised to write in the specific catalogue instance, which represents a subset of the namespace. If not, the request is not redirected.

Similarly, when read requests are also secure (configurable per catalogue instance, as described in the next section), the naming service can also not redirect a request to a catalogue instance, effectively hiding those datasets from some users or groups. This implements the requirement of having private data(sets) in the system.

A disadvantage of the HTTP-based redirection schema is that a single request cannot span multiple catalogues. For instance, it is not possible to list both `mc` and `user` datasets in the same request, because these are contained in different dataset catalogue instances. In practice, these queries are usually not necessary and can be implemented by two separate client requests. Partitioning very large queries is also desirable, as to avoid overloading server resources.

5.2.4 Security

The primary dataset catalogue interface implements a security layer based on the Grid Security Infrastructure (GSI) [76]. GSI is a public key cryptography system (also known as asymmetric cryptography). GSI includes a specification for secure communication between software components that provides authentication and delegation. It relies on certificates that contain information used to identify and authenticate the users or services in the system. A third-party Certificate Authority is used to certify the link between the public key and the subject in the certificate.

The application code running on the server uses GSI to obtain the user identification from the secure HTTP request. For instance, when the user creates a dataset using the method `create_dataset`, the user identification is obtained from the certificate and stored as the dataset Owner attribute.

For performance reasons, managers may disable the use of GSI for read requests. This configuration can be set per catalogue instance. Public key cryptography systems involve computationally expensive operations. Because it is expected that read operations are more common than writes, disabling GSI for read requests results in increased server stability and allows a server to handle a higher number of simultaneous requests. Write requests always require the use of GSI because of the need to obtain user identification.

5.2.5 Fault Tolerance and Scalability Properties

This section reviews fault tolerance and scalability properties in the dataset catalogue design.

Fault tolerance. The naming service is a centralised service, which can be duplicated into multiple instances provided that all naming service instances share the same set of rules. This replication avoids a single point of failure, at the cost of having to stop all instances when rules need to be altered. (It is assumed that such configuration changes are very rare.) In addition, in the event of failure of a dataset catalogue instance, the partitioning of the dataset catalogue into multiple independent instances ensures that only a part of the dataset definitions becomes unavailable, which is an improvement to a single point of failure. The final comment is the protection against data loss within the relational database, which is ensured by the ORACLE relational database system that contains several built-in data protection mechanisms.

Scalability. The partitioning of the dataset catalogue into separate instances ensures that the dataset catalogue can cope with a higher number of entries, provided that the namespace can be adequately partitioned. In addition, the usage of HTTP functionality allows for caching and streaming of (large) replies. In particular, it is expected that most datasets are eventually frozen, which allows for more aggressive caching. Also, storing the differences in files between consecutive dataset versions results in reduced bookkeeping needs, hence improving the overall scalability. Finally, the ability to enable or disable GSI also improves the overall server performance and the ability to serve a higher number of simultaneous requests.

5.3 Storage Services

Storage Services are the local components in the architecture responsible for interacting with the storage at each data centre, to transfer, delete and lookup files. There is usually a single storage service instance deployed per storage, where it runs as a background process serving requests from the dataset masters. The administrators can configure the storage services to serve all or a subset of the dataset masters. (The default is to serve all dataset masters.) Having a storage service instance serve only a subset of the dataset masters can be used to partition the system vertically into separate instances (e.g. to deploy two independent storage service instances, each serving requests from a different master), or to prevent requests from a specific master from being served.

The storage services interact primarily with the dataset masters and local storages, but also make use of the fabric information service. In particular, storage services are able to detect dynamically that a new dataset master instance has been deployed in replacement of a previous instance. This ability is deemed important in a distributed environment where coordinated downtimes are undesirable. Section 5.3.2 ("Lookup Service") details this mechanism, although it applies also to the transfer and deletion services. Nonetheless, this mechanism is limited to the discovery of a new dataset master instance that replaces an existing instance. It does not detect new instances automatically, except if these replace an older instance. Because such deployment changes are expected to be rare and must be coordinated (e.g. the storage administrator must decide if the storage should serve requests for the new dataset master), these changes require a manual reconfiguration of the storage services.

Before describing the design of the storage services, I first describe the interface used to interact with the storage systems. This is followed by the description of the lookup and delete services. The transfer service, which is the most complex storage service, is then presented. This is followed by a discussion on security considerations. While some topics on fault tolerance and scalability are presented throughout the next sections, the main discussion is postponed to the Dataset Master section where the interactions between storage services and dataset masters are analysed in Section 5.4.5.

5.3.1 Storage Interface

An underlying design principle in the distributed data management system is the ability to support heterogeneous storage systems using a layered approach. Following this principle, the system uses a common mass storage interface that is implemented by several storage vendors, called SRM [157]. SRM, which has been the result of an international collaboration and is published as an Open Grid Forum[8] (OGF) specification, allows the storage services to interact with the associated storage system, as shown in Figure 5.4.

[8] Refer to *http://www.ogf.org*.

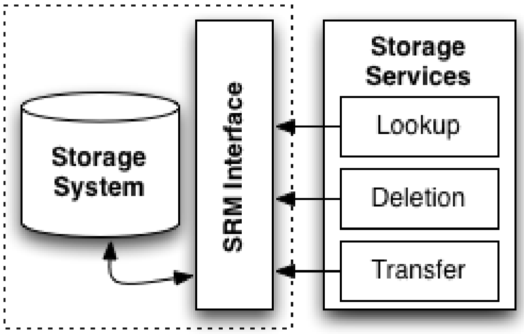


Figure 5.4: Overview of storage system, SRM and storage services.

The goal of SRM is to provide a common storage service with an interface to storage resources, as well as advanced functionality such as dynamic space allocation and file management on shared storage systems. Although storage vendors provide systems with significantly different functionality, SRM attempts to bridge these differences by providing a minimal, inter-operable interface.

In SRM, files may be `ONLINE`, `NEARLINE`, `ONLINE_AND_NEARLINE` or `OFFLINE`. A file `ONLINE` has the lowest latency possible. No further latency improvements are applied to online files. A `NEARLINE` file can have its latency improved to online latency automatically by staging the file to online cache. `OFFLINE` files need human intervention to achieve online latency. These concepts can be mapped to the discussion in Section 4.2.3.1, in particular to Figure 4.2.

Two important concepts in SRM are the SURL and the TURL. The SURL, or site URL, is an abstraction of the file namespace, in the form `srm://ibm.cnaf.infn.it:8443/test`, where `ibm.cnaf.infn.it:8443` is the SRM server host and port, and `/test` is the file path. When requesting a file from an SRM, an SURL is provided. The storage can have the file in several locations (e.g. several disk servers or tapes) and may need to bring it from tape to be accessed. When this is done, a TURL, or Transfer URL, is returned to the user. The TURL includes file access protocol information. For the previous example, the TURL could be `http://srv01.ibm.cnaf.infn.it/test.copy1`. Note that the TURL now includes an HTTP access protocol[9] and a different file path.

TURLs have a lifetime, which is independent of the file's lifetime in the storage (i.e. of the SURL lifetime). For instance, if a file is brought from tape to disk, the lifetime of the disk copy is usually short. After the file has been accessed, that disk copy is removed to give space for other tape recalls. The lifetime of the disk copy is the TURL lifetime, while the lifetime of the file on the storage is the SURL lifetime.

Therefore, the SURL lifetime is chosen by the user (i.e. until the user decides to delete the file), while TURL lifetimes are primarily chosen by the storage. TURL lifetimes

[9] Additional protocols are supported besides HTTP. These are described in a later section.

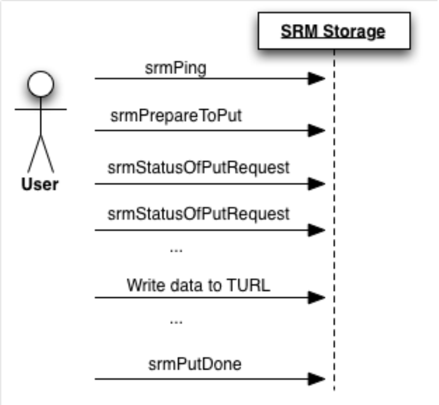


Figure 5.5: Interactions when writing a file to SRM.

and additional copies do not apply only if there is a tape backend. Even when there is no tape backend, the storage may temporarily copy the file to another disk server from where it will be read, improving the read performance. This additional copy is identified by a TURL, and has its own lifetime.

The SRM implementation relies on asynchronous operations. The full listing of SRM methods used by the storage services is included in Appendix B. Figure 5.5 shows an example of the SRM methods required to write a file to the storage. The first interaction is `srmPing`, to check if the service is operational. Then, the user requests a TURL onto which to write the new file, using `srmPrepareToPut`. The user occasionally polls the SRM to see if the TURL is available, using `srmStatusOfPutRequest`. When the TURL is available (i.e. when the storage space is allocated, and path and namespace entries created), the user starts writing the data onto the provided TURL using a transfer protocol supported by the storage. When the write operation is finished, the user executes `srmPutDone`.

Having described the storage interface, I now describe each of the storage services components, and their usage of this interface.

5.3.2 Lookup Service

The lookup service is the component of the storage services used by the dataset master to find files in a storage. Using the lookup service, the dataset master can provide advanced functionality. For instance, after a user has created a dataset, the dataset master requests the lookup service to check if the constituent files are actually present on the storage. This serves as a consistency check for data recently uploaded to a storage. Similarly, when there is a suspicion of lost or corrupted data, the dataset master can request the lookup service to determine whether the system attributes for the files are consistent, by comparing file sizes and checksums. In addition, when the dataset master

```
while True:
    try:
        request = master.get_lookup_job(storage)
    except MasterCommunicationError:
        master = information_service.refresh(master)
        continue

    surls = resolve_surls(request)
    try:
        reply = srm.srmLs(surls)
    except SRMError:
        reply = get_srm_error()

    reply_sent = False
    while not reply_sent:
        try:
            master.set_files_result(reply)
            reply_sent = True
        except MasterCommunicationError:
            master = information_service.refresh(master)
```

Listing 5.2: Pseudo-code for the lookup service.

```
bcbf5ecf-ee2b-4ac1-8710-1c6df66572f3
6bcb3554-5976-40bc-beba-74b977aac342
```

Listing 5.3: Example of a dataset master lookup request.

needs to find appropriate sources to replicate a file, it requests the lookup service to verify whether the files are available at some desired source storage.

At start-up, the storage services spawn one or more lookup service instances for each dataset master being served. The number of lookup service instances per dataset master is configured by the storage administrators to ensure prompt lookup performance without overloading local storage resources, and in particular the SRM interface. Each lookup service instance executes the pseudo-code shown in Listing 5.2. It consists of a loop that requests and processes requests (lookup jobs) from a single dataset master. There is no local checkpoint: if the lookup service crashes, a new instance is restarted. The dataset master is responsible for handling the crashed request using mechanisms described in Section 5.4.5.

The dataset master request includes the list of GUIDs to lookup as shown in Listing 5.3. There are two steps in the lookup procedure. The first is to determine the SRM SURL from the GUIDs in the request (the `resolve_surls` method in the pseudo-code). The second step is to actually perform the lookup in the storage (the `srmLs` method).

The first step requires a translation from a GUID to a SURL. There are two alternative mechanisms to do this translation: using a translation function or a translation catalogue. In the first case, a function uniquely and statically determines SURLs from GUIDs. This relies on an artifact from the GUID generation process. Even though GUIDs are pseudo-random strings, it is possible to determine their creation date from the GUID string. This is shown in the example reply given in Listing 5.4, where for the first GUID the path `/2008/10/02/` is formed from the GUID creation date (i.e.

```
bcbf5ecf-ee2b-4ac1-8710-1c6df66572f3
  srm://host.name/data/2008/10/02/bcbf5ecf-ee2b-4ac1-8710-1c6df66572f3
  10485760
  AD:0015ff0c
6bcb3554-5976-40bc-beba-74b977aac342
  srm://host.name/atlas/production/run2/my.large.file
  1258291200
  AD:0002b1fe
```

Listing 5.4: Example of a lookup service reply.

`/year/month/day/`). The remaining parts of the path are assumed to be static. This simple mechanism can provide adequate distribution of the files in sub-directories[10]. Nonetheless, storage administrators can implement alternative translation rules because the storage services are deployed close to the storage and managed by the storage administrators.

The second alternative is based on a translation catalogue, also known as a *local file catalogue*[11] or LFC. This results in a directory hierarchy that is more human-friendly and consequently easier to manage. An example of the resulting translation using this technique is shown for the second file in the example of Listing 5.4. The LFC is a specification for a service that stores a mapping between GUIDs and SURLs in a relational database. There are multiple implementations of local file catalogues that are supported by the storage services, such as the LCG LFC [19] and the Globus RLS service [42]. These implementations provide advanced functionality and additional flexibility as compared to the translation function. For instance, storage administrators can re-structure the namespace easily, by updating the storage and catalogue a file or directory at a time.

As a result, the LFC provides a highly efficient storage namespace on top of the actual storage namespace. While this enables advanced functionality like quota overviews, which traditional storage namespaces usually do not efficiently support, it introduces additional consistency issues, as the storage and the LFC can become de-synchronised. For instance, a file on the LFC may not exist on the storage, or vice-verse. Some of these consistencies can be detected by the lookup service and corrected as part of the `resolve_surls` method, while others can simply be logged and fixed by human intervention.

The final step of the lookup service is to check whether the SURL really exists on the storage and whether it is accessible. This uses an SRM interface method, which is the `srmLs` method. The method returns the file size and checksum attributes, which are automatically computed by most storage vendors when the file is first written into the storage.

After both lookup steps are completed, the reply is composed and sent to the dataset

[10] The number of files in a storage directory is an important constraint imposed by storage vendors. Storage administrators must ensure that no directories are overloaded by a large number of files.

[11] Another designation in the literature is “Local Replica Catalogue”.

master (the `set_files_result` method in the pseudo-code). The reply includes the requested GUID and for each GUID, the SRM SURL, file size and checksum. The `AD` in the checksum indicates that the checksum type is ADLER-32 (for information on the usage of ADLER-32 refer to Section 5.3.4.3). In case of lookup errors, the reply includes the error description.

Finally, as shown in the pseudo-code, the requests and replies are sent repeatedly until the dataset master is available and accepts it. As discussed in Section 5.4.5, the dataset master is designed to accept replies for requests it did not perform. After a failure contacting the dataset master, the information service is refreshed (the `refresh` method in the pseudo-code), and a new dataset master instance (at a different address) may be chosen to serve the same request (i.e. a new master takes over the previous master). This mechanism allows dynamic deployment of new dataset master instances without manual intervention at the storage services. (This mechanism is also discussed as part of information service in Section 5.5.2.)

5.3.3 Deletion Service

The deletion service is used by the dataset master to delete files from a storage. It works similarly to the lookup service and shares the same lookup mechanism, using either a translation function or a catalogue to determine the SURLs from the list GUIDs in the dataset master request. The difference is that instead of looking up files in the local storage with `srmLs`, it executes the `srmRm` method to delete the files. In some cases, storages implement asynchronous deletion. The `srmRm` request is internally queued by the storage and executed at a later point. For this reason, the deletion service also checks with the `srmLs` method if the request has been processed and only returns to the dataset master when the request has been completed.

5.3.4 Transfer Service

The transfer service is used by the dataset master to replicate files to a storage. The transfers are executed by the storage services at the destination storage, which pull data from a source chosen by the dataset master. (The mechanism for choosing sources is discussed as part of the dataset master in Section 5.4.2.) The transfer service is also responsible for ensuring the proper use of the underlying fabric, in particular to prevent storages from being overloaded with transfer requests. Unlike the lookup and deletion services, which serve a single request at a time per instance and use a reduced number of instances, the transfer services need to serve a large number of parallel transfer requests to compensate for network latency.

Before introducing and motivating the transfer service design, the first two sections discuss the underlying fabric middleware. The first section discusses the protocols and tools for file transfer. This is followed by the mechanism used to guarantee that storages are never overloaded with an excessive number of parallel requests. This mechanism is based on the concept of transfer channels. The last section introduces the design of the transfer service and its usage of transfer protocols, tools and channels.

5.3.4.1 Transfer Tools

There are a wide variety of protocols available to transfer files between storages. These protocols are implemented by several transfer tools, and several of these tools are supported in the implementation as discussed in Section 5.3.4.3. In this section, I describe some of the available transfer protocols and tools.

To transfer data between two storages located in the same data centre, storage systems often provide various proprietary protocols. For instance, the dCache and CASTOR storage systems provide specific clients and custom protocols for data transfer. These are, respectively, dCap or dCache Access Protocol, and RFIO or Remote File Input/Output. In addition, POSIX I/O is usually supported. These protocols are primarily designed for a local area (trusted) network environment, and do not typically implement security mechanisms that are necessary for data transfers across (untrusted) wide-area environments.

Given two storages in different data centres, which is the main use case in this work, GridFTP is the most logical choice for the transfer protocol. GridFTP is widely supported by the storage vendors. It allows for third-party transfers, which is the ability to initiate, control and transfer files between two storages directly, without any data flowing through the client that requested the transfer. This is a mandatory requirement given the large amounts of data being transferred. There are additional features supported by GridFTP and of interest, such as the ability to resume failed transfers or calculate checksums for newly written files.

There are multiple GridFTP clients available, so there is no particular need to implement one directly in the transfer services. In this implementation, I opted for a GridFTP client that provides more advanced functionality. This is the gLite File Transfer Service, or FTS [103]. FTS provides support for GridFTP clients but also supports SRM. As such, instead of specifying TURLs[12], FTS accepts SURLs directly as input and does the necessary SRM calls to determine the TURLs (i.e. referring to the methods described in Appendix B, these are `srmPrepareToGet`, `srmPrepareToPut`, etc). In addition, FTS includes internal retry mechanisms for failed GridFTP or SRM calls, making the system more robust against failures. FTS also implements an additional functionality for preventing storages from being overloaded. This is the subject of the next section.

[12] That is, Transfer URLs in the form `gsiftp://srm://source.host/data/file1`.

5.3.4.2 Transfer Channels

Storage systems (mostly) implement mechanisms to avoid failures due to high load of requests. For instance, if a large number of accesses occur in a short period of time, a storage may refuse some of these requests. Nonetheless, this protection against denial-of-service attacks is usually not sufficient to guarantee adequate transfer performance. In Chapter 6, a detailed analysis of transfer performance is presented and one of the main conclusions is that the number of parallel reads and writes to a storage is a major factor in the overall file transfer performance. While a high number of parallel requests may not resemble a denial-of-service attack, it may be sufficient to significantly affect the storage performance as shown in Chapter 6.

In addition, storage systems may be used simultaneously by different organisations: using Grid terminology, by different Virtual Organisations (VOs). Therefore, it is desirable to have a common system that throttles the transfers between storages on the wide-area network, ensuring that adequate performance is maintained overall. This functionality is provided by FTS, and its throttling mechanism is based on the idea of *transfer channels*. A transfer channel is a virtual unidirectional link between a source and a destination storage. For instance, the channel `CERN-BNL` serves the transfer of files from CERN to the Brookhaven National Laboratory (BNL) for all virtual organisations. Transfer channels were first introduced as part of FTS, but their usage and scope has been augmented in this implementation. In this section, I describe the basic FTS model. (The extensions are described in the next section.)

A channel in FTS has a number of transfer slots per virtual organisation and a set of channel configuration settings. The transfer slots are the maximum number of files in transfer at any point in time for a virtual organisation. If the channel `CERN-BNL` has 10 transfer slots, then at most 10 files can be in transfer from CERN to BNL by the virtual organisation. This implies at most 10 read and write requests in BNL and CERN respectively imposed by the channel.

The channel configuration settings allow the adjustment of several settings in the underlying transfer protocol. For instance, since CERN is located in Switzerland and BNL in the United States, the GridFTP transfer buffer size is set to a slightly higher value than for other channels whose source is in Europe, as to compensate for the higher network round trip time. The storage administrators at CERN and BNL can negotiate the best channel settings based on network considerations. In addition, some channel configurations can be set dynamically by specifying a range that is used for automatically adjusting buffer sizes depending on file sizes.

The gLite FTS service implements this transfer model. The FTS service receives and executes transfer jobs. A transfer job consists of a list of transfer requests. Each transfer request specifies the source file and the destination file path. The FTS server parses

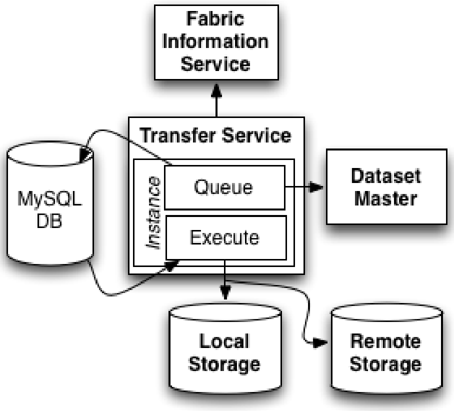


Figure 5.6: Overview of the transfer service.

the job and inserts each transfer request into a first-in-first-out (FIFO) queue for the corresponding channel. For instance, requests in the form `srm://cern.ch/data/file1` `srm://bnl.gov/data/file1` are allocated to the `CERN-BNL` channel by parsing the host names from the SURLs.

Asynchronously, FTS transfer agents pick up transfer requests from each channel's FIFO queue and execute the transfer. The number of active transfer agents for a channel corresponds to the number of transfer slots for the channel (i.e. the number of simultaneous transfers). As such, the gLite FTS server is essentially a wrapper around the transfer protocol that implements the transfer channel model, hereby increasing the overall stability of transfers by preventing storages from being overloaded.

Nonetheless, the FTS model has an important short-coming. It has limited support for dynamic scheduling of transfers, because requests are allocated onto a first-in-first-out queue and can at most be raised in priority after being queued. The next section describes how the basic FTS model has been expanded in the transfer services with support for fair sharing and just-in-time scheduling.

5.3.4.3 Design

Figure 5.6 describes the design of the transfer service. At start-up, the storage services spawn a single instance of a transfer service for each dataset master being served. The instance contains two components: a queueing component that requests transfers from the dataset master and queues them into a local database, and an execution component that reads queued requests from the local database and executes the transfers.

The partitioning of transfer services into separate queueing and execution components is justified by the need to handle a large number of requests, while ensuring that service failures do not always force requests to be retried. With a local database, it is possible

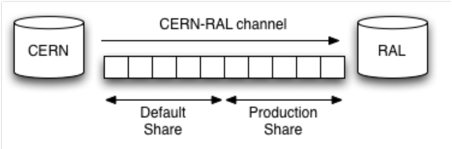


Figure 5.7: Example of fair shares.

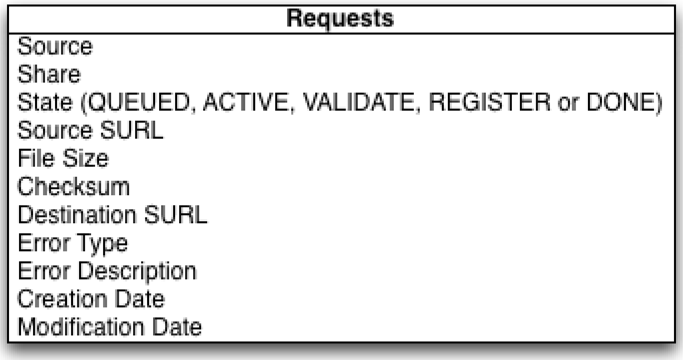
```
c0cfe446-f8e9-4423-bfed-997446852816 srm://source.host/file1 10485760 AD:0000ff01
16c0bbfa-e0f1-42e5-840f-654654138e1a srm://source.host/file2 10485760 AD:0000af12
05bfec9a-a0e8-49fb-b22e-10c81a2d3826 srm://source.host/file3 10485760 AD:0000cff3
50fab37a-b956-4ccf-ba5c-58dd65af5a5e srm://source.host/file4 10485760 AD:0000df24
1321c16b-f064-4515-a804-bcb4793fc9db srm://source.host/file5 10485760 AD:0000efb5
8d93e6cb-82b1-495e-b55f-8557261c0a00 srm://source.host/file6 10485760 AD:0000ffc6
```

Listing 5.5: Example of a dataset master transfer request.

in some situations for the system to recover its state from the local database without retries by the dataset master. (The occasions where state can be recovered are discussed later.) Unlike the lookup and deletion services, the transfers can take a long time (e.g. 20 minutes is not uncommon) hence increasing the cost of a failure and its retrial. Additionally, the lookup and deletion requests can be safely retried without significantly overloading the system or disrupting other components. On the other hand, transfer requests always involve two parties (source and destination storage) so these must be handled with greater care and (unnecessary) retrials should be avoided.

The previous section describes the model developed in the context of FTS to structure the system into virtual transfer channels each with a specific number of transfer slots. This model has been adopted and expanded in the transfer services. The modification consists in the introduction of *fair shares* for each transfer request. An example of fair shares is shown in Figure 5.7. The figure illustrates a single channel (from `CERN` to `RAL`) as in FTS, but now with two shares dividing the available transfer slots in half.

Fair shares serve to allocate quotas (of transfer slots) within each channel. When a user requests a dataset subscription in the dataset master, the request includes the assigned share. The request is later fulfilled by the transfer service using a specific transfer channel and its share allocation. Fair shares allow for dynamic scheduling of transfers and good resource usage: if a share is not used by any request, then other shares are allowed to use its transfer slots, leading to better transfer performance; if a request is assigned to a share, it is guaranteed a subset of the available transfer slots (as soon as any ongoing transfers are complete). An alternative system based on priorities could cause starvation with lower priority requests never served if there is a long queue of high priority requests. In addition, while the set of available shares is configured centrally in the information service, the quota of each share is configured locally, allowing local administrators to control the inbound traffic imposed by different activities (which are mapped to different shares).



Requests

Source

Share

State (QUEUED, ACTIVE, VALIDATE, REGISTER or DONE)

Source SURL

File Size

Checksum

Destination SURL

Error Type

Error Description

Creation Date

Modification Date

FIGURE 5.8: Schema of the transfer service.

```
sources = information_service.read_channels()
shares = information_service.read_shares()
quotas = configuration.read_quotas(shares)
threshold = configuration.read_thresholds(sources, shares)

while True:
    for source in sources:
        for share in shares:
            if mysql.number_queued_requests(source, share) < threshold[source][share]:
                request = master.get_transfer_job(source, storage, quotas[share])
                mysql.insert_request(request)

                done = mysql.get_done_requests(source, quotas[share])
                if done:
                    master.set_files_result(done)

                queued = mysql.get_queued_requests(source, quotas[share])
                if queued:
                    master.set_files_result(queued)
```

LISTING 5.6: Pseudo-code for the queueing component of the transfer service (error-handling not included).

The local database in the transfer service is based on MySQL[13]. MySQL was chosen for its straightforward deployment, which is an important factor given that the storage services are deployed at several data centres. Its schema is shown on Figure 5.8. The schema stores the transfer requests and their `State`, which is `QUEUED` for newly inserted transfers, `ACTIVE` for ongoing transfers, `VALIDATE` for completed transfers not yet validated, `REGISTER` for completed transfers not yet registered (registration is discussed later) or `DONE` for either registered transfers or failed transfers. An example of a transfer request received from the dataset master in given in Listing 5.5, where each line contains the file GUID, source SURL, file size and checksum.

Listing 5.6 describes the queueing component of the transfer service. At start-up, the channels and shares are read from the information service. The local quota per share is read from a local configuration file. From there on, the queueing component loops through all channels and shares, maintaining a buffer of transfers based on a set of

[13] Refer to *http://www.mysql.org*.

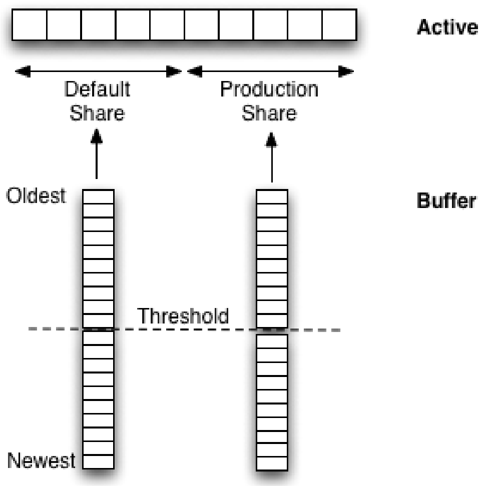


FIGURE 5.9: Fair shares and buffers.

thresholds per channel and share. Whenever the threshold is passed, the queueing components contact the dataset master and ask for additional transfers for this channel and share. This model allows the transfer services to maintain a buffer of transfers, which compensates for the slowness in contacting the dataset master. Provided that this buffer is sufficiently small, the model allows the transfer service to send requests promptly to the transfer tool, enabling close to just-in-time scheduling.

This model is illustrated in Figure 5.9. In the example, there are 10 transfer slots with 2 shares, each with half the available slots: these are shown as the `Active` slots. Each buffer per share has a threshold of 10 files to transfer (shown as `Buffer` slots). Whenever the threshold is passed, the queueing component requests an additional 10 transfers: these slots are shown in the bottom part of the figure under the threshold mark. In this scenario there are usually between 20 to a maximum of 40 transfers always in the buffer. Any share has enough transfers in the buffer to fill the entire `Active` queue, in case the other share does not have any requests queued. Nonetheless, too large buffers prevent the dataset master from having a more dynamic scheduling of transfers, because the dataset master could have assigned the transfer to another suitable source with a smaller buffer.

As shown in Listing 5.6, the queueing component is also responsible for replying to the dataset master when requests are completed. (The queueing component reads these requests, in state `DONE`, using the `get_done_requests` method.) In addition, the queueing component must also notify the dataset master regularly on its queued requests. This is a requirement of the lease mechanism implemented between the dataset master and transfer services, and is described in Section 5.4.5. (The renewal is shown in the pseudocode as the `get_queued_requests` method and the corresponding `set_files_result` method.)

```
c0cfe446-f8e9-4423-bfed-997446852816 srm://destination.host/data/file1
  OK
16c0bbfa-e0f1-42e5-840f-654654138e1a srm://destination.host/data/file2
  TEMPFAIL
  'GridFTP read error'
05bfec9a-a0e8-49fb-b22e-10c81a2d3826 srm://destination.host/data/file3
  OK
50fab37a-b956-4ccf-ba5c-58dd65af5a5e srm://destination.host/data/file4
  PERMFAIL
  'Source file not found'
```

Listing 5.7: Example of a dataset master transfer reply.

```
tool = configuration.get_transfer_tool(source)

while True:
    request = mysql.get_next_queued_request()
    dest_surl = choose_destination_path(request)

    attempts = 0
    done = False
    while attempts < max_attempts and not done:
        result = tool.transfer(request, dest_surl)
        if transfer_success(result):
            mysql.set_state_validate(request)
            if tool.validate(dest_surl):
                mysql.set_state_register(request)
                register_surl(request)
                done = True
            else:
                tool.delete(dest_surl)
        else:
            tool.delete(dest_surl)
            if is_error_permanent(result):
                done = True

    mysql.set_state_done(request, result)
```

Listing 5.8: Pseudo-code for the execution component of the transfer service (error-handling not included).

An example of a reply sent to the dataset master for `DONE` files is shown in Listing 5.7. The reply illustrates the three types of states for a completed transfer. `OK` is a successfully completed transfer. `TEMPFAIL` is a transfer that failed with what is considered a temporary fault; as such, it may be retried later and will likely succeed. `PERMFAIL` is a transfer that failed with what is considered a more serious, potentially permanent fault, which will (likely) not succeed by retrying. In the example, the source file was not found, so no attempt to transfer from the same source should succeed. The reason that sometimes even permanent failures succeed on retry is due to bad error reporting or more complex error conditions from the storages. For instance, a disk server may be taken offline for repair and some storages report the data as lost, while others report the event as a temporary fault: if the disk server is restored later with all data, these storages will report the data as available. These two error classes are used by the dataset master in its retry mechanism.

Listing 5.8 describes the (simplified) pseudo-code for the execution component, which is responsible for executing the transfers. At start-up, the component chooses the transfer tool to use for the channel, based on the source and destination storages. The available tools are configured locally and may be e.g. a local `cp` command if both storages are within the same network domain or the FTS GridFTP client for wide-area transfers. The component then enters the loop that reads requests from the local MySQL database (the `get_next_queued_request` method). For each request, it composes the destination path based on local storage conventions (implemented in the `choose_destination_path` method). The transfers are then executed (using the `transfer` method) and validated if successful (by the `validate` method). Failed transfers are deleted (shown as the `delete` method) and successful transfers may require a registration. The registration is required if the storage services rely on a local file catalogue, as described in the lookup service in Section 5.3.2.

Completed transfers are validated by comparing the newly written file with the file size and the checksum provided by the dataset master. After a transfer is completed, the transfer tool reports back the file size and the checksum of the newly written file. ADLER-32 [63] is the default checksum used in the system since it is widely supported by storage vendors. This support is due in part to its rolling hash property, which allows the checksum to be computed as the input moves through a window, i.e. as the file is written to disk. This eases the checksum computation without introducing significant overheads. (Tape drives also often compute ADLER-32 at the hardware level when writing files to tape, which provides another verification step.)

Whenever a failed transfer is reported by the transfer tool, the execution component does not immediately report back the error. Instead, as shown in the pseudo-code, it will internally retry the transfer a configurable number of times and only report back the last error if the transfer fails for all attempts. This only applies if the tool does not consider the error to be of a permanent nature (this is verified in the `is_error_permanent` method). The goal is to avoid a large number of exchanged messages between the transfer services and the dataset master. This could cause the dataset master to be overloaded with messages particularly during failures of short duration.

Listing 5.8 also describes the state changes recorded in the MySQL database. This allows recovery of some requests in the case of an unexpected crash. For instance, if a file has been transferred and a crash occurs after the MySQL database state changes to `REGISTER` but before being set to `DONE`, the request can recovered without re-copying the file but only with a re-registration attempt. A similar situation occurs between the `VALIDATE` and `REGISTER` states. Nonetheless, it is always possible that transfer services crash half-way during a transfer, causing partially written files to be left in the storage system. The SRM layer is capable of detecting and correcting these events, because the last SRM call (`srmPutDone`) is never executed and times out, leading the SRM to remove the leftover data.

5.3.5 Security

In the previous sections, the assumption has been that the communication between dataset masters and storage services is secure. There are two alternative mechanisms to establish this secure environment. The first is to use GSI and assign a service certificate to both services. This establishes a secure communication channel between both trusted parties. Another possibility, which does not implement true secure communication but is sufficient for some situations, is to use firewall configurations and IP-based white lists. This has the advantage of avoiding the overhead imposed by the GSI connections at the cost of having administrators setup the required network firewall configurations manually[14]. This latter option does not require any changes to the storage services or to the dataset master.

5.4 Dataset Master

The dataset master is the global service in the system that implements the data distribution functionality. As described in Chapter 4, there may be more than one dataset master. This form of vertical partitioning is enabled by a redirection service described in this section.

The dataset master receives dataset subscription requests from users and interacts with both the dataset catalogue and the storage services to fulfil the user requests. While the dataset catalogue interface contains methods to manipulate logical definitions, such as datasets or logical file names, the dataset master contains methods to manipulate the physical instantiation of these definitions, which are the replicas of the datasets. This implements the split between logical and physical units discussed in Section 4.4.2.2.

The first section describes the schema of the dataset master. This provides the background for the second section, which describes each of the dataset master methods and corresponding implementations. This section includes both the methods used between the dataset master and the users, and methods used between the dataset master and the storage services. The third section describes the redirection service, and the last two sections discuss security and review fault tolerance and scalability properties. In particular, the fault tolerance discussion addresses important issues in the interaction between dataset masters and storage services.

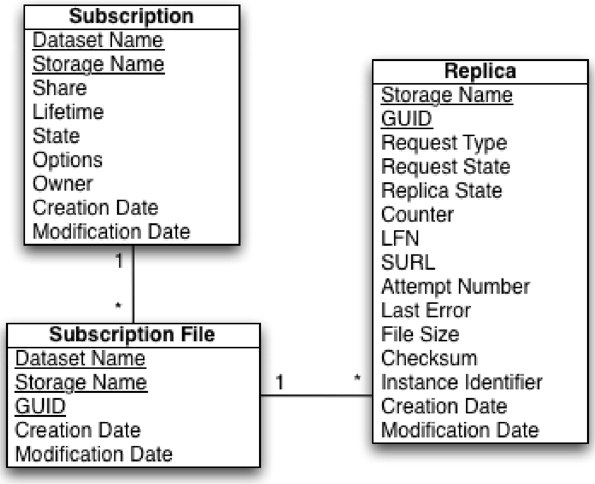


FIGURE 5.10: Schema of the dataset master (primary keys are underlined).

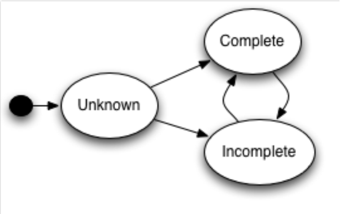


FIGURE 5.11: States of a subscription.

5.4.1 Schema

Figure 5.10 shows the schema of the dataset master[15]. `Subscription` contains the subscriptions for a storage. Each storage is identified by a human-readable `Storage Name`. `State` stores the current state of the subscription, which is one of the states shown in Figure 5.11. `Unknown` is the initial state that is set before the system checks whether the constituent files are present or missing at the storage. `Incomplete` is the state set when only some subset of the files in the dataset are available, and `Complete` when all files are available. The schema includes a many-to-many relationship between `Subscription` and `Replica`. `Replica` stores the files available at a storage. The many-to-many relationship relies on the intermediary `Subscription File`. This follows from the fact that files may be part of multiple datasets. Therefore, a (subscribed) file may be part of many (subscribed) datasets.

The `Replica` includes a `Request Type`, `Request State` and a `Replica State`. I describe each of these in turn. (The next section also describes these state transitions in

[14] In fact, GSI also has specific firewall requirements since it needs a TCP port range to be configured, but this is a simpler configuration that IP-based while listing or other lower level routing protections.

[15] The schema presented in this section is a simplified view of the actual schema for readability purposes. It includes the primary entities but leaves out the quotas, users, groups and notifications, which are secondary issues to the discussion.

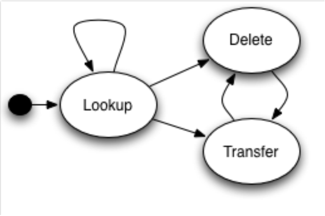


Figure 5.12: Types of a file request.

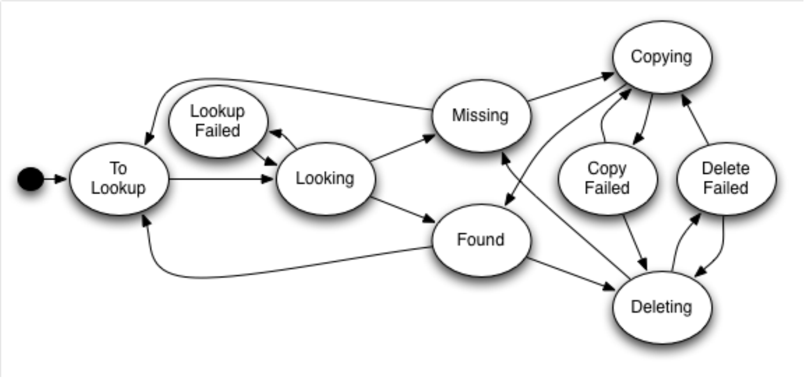


Figure 5.13: States of a file request.

detail.)

The request types are shown in Figure 5.12 and correspond to the tasks performed by storage services: `Lookup`, `Transfer` and `Delete`. Whenever the storage service asks for a specific task (e.g. `Lookup`), only entries in `Replica` with the corresponding request type are returned.

In addition, each request has a state. These are shown in Figure 5.13. For instance, a file with request type `Lookup`, is either in state `To Lookup` (i.e. ready to be picked up by the storage services), `Looking` (i.e. picked up by the storage services and in processing), or `Lookup Failed` (i.e. the storage services processed the request but failed with an error). After the request is processed successfully by the storage services, the request state is set to `Found` or `Missing`. This applies for lookup, transfer or deletion requests (e.g. `Missing` may be due to a lookup request not finding the file or to the successful deletion of an existing replica).

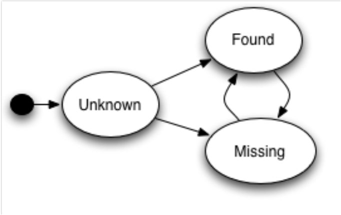


Figure 5.14: States of a file replica.

Also, each replica has a state. This is recorded in the Replica State field, which is a redundant field. Its states are shown in Figure 5.14. The Replica State corresponds to the last known final request state. For instance, if at some point the request state is set to Found, the Replica State changes to Found. This way, even if the request state changes later (e.g. to lookup the file again), the replica state still records the last known stable state, which can be queried by users. The state Unknown is the first state, which is used when a dataset is first registered at a location. At this point, the system does not know whether the file is actually present or not.

One important schema optimisation concerns the choice of database indexes. These are not shown in the schema except for the primary keys, which are always indexed. The Request State entries with values Found and Missing are not indexed, except for the combination Request Type = Transfer $\bigwedge$ Request State = Missing and Request Type = Delete $\bigwedge$ Request State = Found. Because it is expected that most replicas are either in state Found or Missing, this optimisation results in a substantial reduction of the total index size.

5.4.2 Interface

In this section, the dataset master interface is described. Similarly to the dataset catalogue, the interface is based on RPC with a custom protocol built atop of the HTTP protocol. For details on the software stack, refer to Section 5.2.2.

This section presents the interface in two parts. The first part describes the methods available to the users, and the second part describes the methods used for communication between the dataset master and the storage services.

5.4.2.1 User Requests

This section describes the functionality available to users.

Register subscription.

The user specifies the dataset name, destination storage, and the fair share. From this point on, and following the subscription principles discussed in Section 4.4.2.2, the system will try to copy any missing files from the dataset to the destination storage.

When the request is received, the dataset master inserts a new entry in Subscription. It then contacts the dataset catalogue to determine the list of constituent files in the dataset. These files are inserted into Subscription File and also into Replica. Nonetheless, some of these files may have been requested at the storage by some other dataset. For the existing entries in Replica, the corresponding Counter field is incremented. The Counter is an integer that counts the number of subscribed datasets that contain a given file. When `Counter` ≥ 1, the file is either available or in copy, because "`Counter` number of subscriptions" require it.

It is also possible that the entry existed but with `Counter` $= 0$. After this request, `Counter` $= 1$. In this case, the `Request Type` is set to `Transfer`. This implies that the dataset master can now assign this file for transfer by the storage services. In the next method, I describe the inverse of this operation.

Unregister subscription.

The user specifies the dataset name and storage name. The method removes the subscription. From this point on, any files available at the storage not part of any other subscribed datasets can be removed.

When the request is received, the dataset master removes the corresponding entry `Subscription`. This also triggers the removal of the entries for the same subscription from `Subscription File`. In addition, and following the reverse of the operations described in **Register subscription**, the `Counter` field is decremented. If `Counter` $= 1$ initially, after this operation `Counter` $= 0$. In this case, the `Request Type` is set to `Delete`. As such, the dataset master can now assign this file for deletion by the storage services.

Refresh subscription.

This method ensures that updates to the dataset definition are propagated to all dataset locations (or replicas). For instance, after a user adds a file to the dataset, the dataset master is only notified of the new contents when this method is called. At that point, the dataset master will instruct all subscribed storages to try and obtain a replica of the new file.

When the request is received, the dataset master queries the dataset catalogue for the constituent files in the dataset. Any files in the dataset but not in `Subscription File` are inserted, while entries no longer in the dataset are removed. This triggers the underlying transfer or deletion of files, so that physical replicas of the dataset reflect its logical definition. If the dataset no longer exists in the dataset catalogue, this method implements the same behaviour as **Unregister subscription**.

List locations.

This method is used to list the storages that contain a given dataset. It is used to find replicas of datasets in the system. The method reads and returns the `Subscription` entries in `State` $=$ **Complete** $\bigvee$ **Incomplete**. The method also supports wildcards as the dataset name.

List location attributes.

The user specifies the dataset name and storage, and the method returns detailed information on the status of this specific dataset replica. The dataset replica status includes the list of available files at the storage, plus several system attributes for each file, such as the date on which its state was last changed by the storage services (i.e. the last time the replica was accessed by the storage services). The method reads and returns the entries in `Replica` and in particular makes use of the `Replica State` and `Modification Date` fields.

5.4.2.2 Storage Service Requests

This section describes the methods used by the storage services to request work from the dataset master. (Examples of the messages exchanged between both services are presented in the storage services discussion in Section 5.3.) The assignment of work by the dataset master is based on a *lease* mechanism. Whenever a storage service requests work (e.g. a lookup request), the request is assigned exclusively to this storage service. During the lease, the storage service is expected to perform some operation on the file (e.g. a lookup operation). When the storage service completes the operation (successfully or failed), it releases the lease, notifying the dataset master of the result. Leases have a duration. If a request is expected to take a long time (e.g. a transfer request may be queued for some time in the internal buffer), then the lease needs to be refreshed. This is the reason why the transfer services also notify the dataset master of the state of queued requests. If the lease expires, the dataset master is allowed to re-assign the request. As discussed later, the dataset master is able to guarantee, under some restrictions, that the system achieves a consistent state even if a lease expires and is re-assigned.

I now describe each of the dataset master methods in more detail.

`get_lookup_job(storage[, n])`.

This method returns a set of at most `n` number of files from `Replica` with `Request Type` = `Lookup` and `Request State` = `To Lookup` $\bigvee$ `Lookup Failed`. It also sets the `Request State` to `Looking`. The requests are restricted to be for the `storage` and are ordered by oldest `Modification Date`.

`get_delete_job(storage[, n])`.

This method returns a set of at most `n` number of files from `Replica` with `Request Type` = `Delete` and `Request State` = `Found` $\bigvee$ `Delete Failed` $\bigvee$ `Copy Failed`. It also sets the `Request State` to `Deleting`. The requests are restricted to be for the `storage` and are ordered by oldest `Modification Date`.

`get_transfer_job(source, destination, share, n)`.

This method returns a set of at most `n` files to be transferred by the destination storage services, for the given channel (from `source` to `destination`) and share. The method implements the data flow in the system, because it assigns the transfers for each channel. It assumes that the network infrastructure uses the channel model described in Section 5.3.4.2. In addition, it assumes that there are channels available between any two storages. That is, using a graph theoretical terminology (refer to e.g. [93]), where the transfer channels are the graph edges and the storages are the graph nodes, the graph must be complete (i.e. for n nodes the graph must be K_n). Finally, the method assumes that the internal buffers in the transfer services are reasonably small and that the buffering time can be ignored, otherwise the transfers could have potentially been assigned to another available channel. Note that no assumption is made on the channel performance. This point is discussed later.

The files that qualify for transfer are those where the destination file has `Request Type` = `Transfer` and `Request State` = `Missing` $\bigvee$ `Transfer Failed` $\bigvee$ `Delete Failed` and the source file has `Request State` = `Found` (i.e. where the destination is missing and is a transfer request, and the source is available). After choosing a set of files to transfer, the `Request State` of the destination file is set to `Copying`. In addition, there are four additional criteria. I describe each of these in turn.

A criteria to choose the set of files to transfer is the "distance" between the storage systems. Close storages, which are described as part of the Fabric Information Service in Section 5.5.2, are a list of storages that ought to be preferred because they are considered to be nearer in network terms. While all storages are connected, the close storages represent preferred sources (e.g. connected through dedicated network links). If the method is called from a `source` close to the given `destination`, the criteria is met. If not, there is an extra validation. The objective of this extra validation is to delay serving file transfers from non-close source storages for some (configurable) period of time with the expectation that some future request for the same destination will be made from a close source. Therefore, this validation consists on checking the `Modification Date` and skipping "recent" requests according to a delay period. After the delay period has elapsed, the criteria is also met and transfers can be served. This mechanism is only appropriate for environments where there are always queues of requests waiting, otherwise there could be periods of time where channels are not used even though requests are available. In addition, this mechanism applies for systems where the datasets are distributed broadly, so that a close source will usually be available or will receive the data shortly after. Both conditions occur in our usage scenarios as discussed in Chapter 6.

There are two additional criteria that apply in the general case, and one criteria that applies for failed transfers. For the general case, the two additional criteria are the age of the request and the dataset completion. The system ensures that older transfer requests ordered by `Modification Date` are served first. This prevents starvation, which could cause some requests never to be served. The other criteria is the contribution of this transfer to the completion of some dataset. Similarly to the principles applied in BitTorrent [48], an important goal is to complete dataset transfers as quickly as possible, particularly when only a few files are missing from the dataset. For instance, if a dataset misses a single file, this transfer is sent first to the storage services. By completing datasets early, the system allows users to continue with their analysis tasks sooner. This may also prevent a data loss from affecting an almost complete transfer. This failure would be of high cost considering that many transfers have already been performed for the dataset.

The final criteria applies for failed transfers (i.e. requests where `Request State` = `Transfer Failed`). It is not desirable to retry transfers immediately. Instead, an exponential back-off is implemented between subsequent attempts. The rationale is the following: a transfer error can happen due to transient failures (e.g. network faults) or more serious service failures. In the case of transient failures, an immediate (or shortly after) retry will likely not suffer from the same failure. Transfer errors that involve a failed service somewhere in the infrastructure have a time-to-recovery. This time-to-recovery is unknown but it is likely that two failed transfers with an interval of, for instance, 1 second will suffer from the same error. On the contrary, transfers with an interval of 1 hour will likely not suffer from the same error. In this case, failures are independent[16]. Therefore, if another failure occurs after a longer retry period, this is either due to a long service failure or to the occurrence of (probabilistic) independent failures. Both these events should (intuitively) occur with much lower probability. The final point is the choice of the retry interval. Because the time-to-recovery is not known, a suitable approach is to exponentially increase the interval between retries. This interval is capped to a maximum, to avoid very long intervals.

Having a variety of sources for a transfer is important. Therefore, whenever the method is unable to serve some requests, it inserts a lookup request for those files in its close source storages. For instance, an incoming request from a destination storage is not served because the only available source is “distant”. Other (more recent) requests may be returned for transfer. Nonetheless, the method inserts requests with `Request Type` = `Lookup` at various close storages for the skipped files. Whenever future requests from the same destination storage arrive, there are (potentially) newly found close sources, which allow the previously skipped files to be assigned for transfer.

Finally, users can configure some filtering to be applied to this algorithm, forbidding or using only a subset of the available sources. This is set by the users at subscription time and stored as `Options` in the `Subscription` entry. The `get_transfer_job` method takes these options into account when choosing sources.

For performance reasons, the `get_transfer_job` method does not make these scheduling

[16] The notion of independence used is from probability theory, where any collection of events A are mutually independent if and only if for any subset of the collection $A_1, \ldots, A_n$, we have $Pr(\bigcap_{i=1}^{n} A_i) = \prod_{i=1}^{n} Pr(A_i)$.

decisions online. The dataset master uses various intermediary buffers implemented in the form of (temporary) relational views, which are refreshed very frequently. Therefore, queries only read and schedule based on a subset of the data in the system.

This transfer algorithm does not attempt to predict transfer performance from past history and schedule accordingly. It only attempts to assign requests towards channels that link close sites. In effect, it implements a *feedback-based model*: channels that are working properly and faster will tend to complete the assigned work sooner and request additional work. Overload conditions are prevented by the introduction of transfer channels that limit the overall number of parallel requests. In Chapter 6 this implementation choice is analysed in greater detail. The reason for not using past history and transfer prediction is that in a worldwide, distributed and shared infrastructure, it is not possible to predict reliably the transfer performance. As shown later, there are multiple factors influencing the transfer performance that cannot be known *a priori*.

`set_files_result(result)`.

This method is used by the storage services to report the result of the lookup, transfer or delete operations on a set of files. It is also used to renew the lease of ongoing transfers, which are buffered in the transfer services. The `result` includes a list of entries where each entry contains the file (identified by GUID), its new state and any additional information required. (Various examples of these messages are shown in the sections that describes the storage services.)

Whenever the `result` includes files that have a new `Request State` $=$ `Found` $\bigvee$ `Missing`, the `Replica State` is set accordingly to either `Found` or `Missing`. In addition, a newly found or missing file may trigger the change of the subscription state. Therefore, all subscriptions that contain these file are scanned to check whether the field `State` in `Subscription` changes to either `Complete` if all files are `Found` or to `Missing` otherwise. With these updates, the dataset master implements a best effort dataset location cache that knows about the replicas in the system, as discussed in Section 4.4.3.3.

As discussed in Section 5.3.4.3, some transfer errors are classified by the storage services as permanent failures. These errors are reported through this method to the dataset master. In this case, the dataset master inserts a lookup request at the source storage to verify that the source is available. (As a side-effect, the dataset location cache in the dataset master is refreshed.) It would be possible to implement more complex models, such as deleting and re-copying the source file from elsewhere, but this mechanism is not implemented. Instead, these errors are logged for administrative monitoring.

5.4.3 Redirection Service

The redirection service is similar to the dataset catalogue naming service described in Section 5.2.3. The objective is to have independent dataset master instances to distribute the request load and achieve better scalability. If there are multiple instances but a single global entity, there is the requirement for a service that uniquely determines a dataset master instance given a dataset name. The redirection service routes user requests to the appropriate dataset master instance using functionality available in HTTP. The routing rules are configured statically by the application managers. These rules depend on dataset names following a well-defined structure, using principles similar to those described in Section 5.2.3.

5.4.4 Security

The security mechanism follows similar implementation principles to the dataset catalogues and storage services. In particular, GSI is used for user write requests while user read requests are not required to be secure. For communication with the storage services, either GSI or network firewall configurations (IP-based) are used. (Refer to Section 5.3.5 for additional details.)

5.4.5 Fault Tolerance and Scalability Properties

This section discusses fault tolerance and scalability properties in the design of the dataset master and in the interactions with the storage services.

Fault tolerance.

In Section 5.3.2, the pseudo-code illustrated the mechanism used by the lookup service to retry the communication with the dataset master in case of failure. Two scenarios were discussed. In the first scenario, the communication fails. In this case, the retrial mechanism within each storage service ensures that the dataset master eventually receives the reply. Below I discuss what happens if this reply is received after the lease has expired. In the second scenario, the dataset master failed and is redeployed at a different location. In this case, the dataset master may receive a reply for a request it did not perform.

In both scenarios, the dataset master ensures that the system eventually reaches a consistent state. The mechanism and a proof sketch follows. I assume a non-byzantine environment [110], where components may fail but behave consistently otherwise; therefore, byzantine fault tolerance mechanisms (such as [40]) are not required[17]. To circum-

[17] Note that the communication between storage services and dataset masters can be made secure by the use of GSI, so that all parties are authenticated.

vent the result by Fischer-Lynch-Paterson [71], which proves the impossibility of solving consensus with any failures in an asynchronous system, I use timeouts and the relative order of messages. I also assume that the delivery of messages can be made reliable (e.g. by recording every message locally before submission and re-submitting the same message after a crash recovery). In addition, each storage service instance has a unique identifier. This is generated when the instance starts and consists of a globally unique identifier. The unique identifier is included as part of the HTTP header in all requests to the dataset master. For instance, the HTTP header of the `get_transfer_job` contains the transfer service instance identifier. Finally, it is assumed that, per storage, no storage service instances share the same dataset master[18].

Let r_k be a job request (e.g. `get_delete_job`) and p_k be the corresponding reply. Whenever a file is sent as part of a request, the `Instance Identifier` field in `Replica` entry is set to the instance identifier. Therefore, it is possible to know that $r_1 \Rightarrow p_1$ correspond to a request and its reply, because both share the same instance identifier. Now suppose a request r_j is sent. The dataset master expects to receive p_j. Suppose instead it receives p_k where $k \neq j$. r_k must be an older request than r_j for otherwise the environment would be byzantine, which is contradictory to the initial assumption. Therefore p_k is the reply to an older request, which can be caused by a delayed processing (due to a slow instance and a timeout, which led the dataset master to re-issue a new request) or by a new dataset master being deployed in replacement of a previous one. In this case, when the dataset master receives p_k, it marks the corresponding file requests as failed. For instance, if at time of r_j the state was set to `Looking`, it is now set to `Lookup Failed`. This implies the file will be part of a future request r_l where $l > j$. Later, when the response to the original p_j arrives, the same process occurs (it is marked as failed and re-sent). But because replies only occur for requests (non-byzantine environment), then clearly at some point there are no more older replies and the last request is sent and received in order, reaching a consistent state. Note that the method also ensures progress because even if p_j never arrives, there is a new request (r_l) already sent.

As an example, suppose the user inserts a subscription. The corresponding files are picked up for transfer but the transfer service is slow. The request times out (no reply is received). In the meantime, the user asked for the deletion of the subscription. Because the initial transfer request timed out, it is now picked up by the deletion service and processed immediately. There are no files to delete (files were not copied) so the request succeeds. Then, the transfer service finally handles the request, copying the files over. At this state, the files were copied but should be missing, because this was the last user request. There are two possibilities. In the first case, the deletion reply arrives before the copy reply. (Previously I mentioned that the deletion was processed successfully but did not mention that the corresponding reply was received by the dataset master.) At first it is accepted but as soon as the copy reply arrives (for an older request), a

[18] As discussed in Section 5.4, separate storage service instances serving the same storage must contact separate dataset masters. This mechanism is used for vertical partitioning.

new deletion is re-sent, reaching a consistent state. In the second case, the copy reply arrives before the delete reply (even though it was executed afterwords). It is marked as failed and a new delete is re-sent. Then, the original delete reply arrives. It is again marked as failed and a new delete is re-sent. Eventually the last delete arrives in order (because the delete service processes a request at a time per storage), leaving the system in consistent state.

Finally, it is possible that requests are never served by any storage service. This occurs when the storage service is not deployed. Monitoring mechanisms (which are not shown in the schema) can detect these conditions by recording the last time a request was processed by a storage service.

Scalability. A potential bottleneck in the interaction between a dataset master and associated storage services is the number of exchanged messages. For instance, if there is a large number of subscriptions inserted in a short period of time, these subscriptions will trigger multiple lookup and transfer requests at various storage services. These messages correspond to the dataset master assigning work to the various storage services and receiving the corresponding replies. In the implementation (and as shown in previous sections containing examples of requests and replies), these messages contain bulk listings of files. Therefore, multiple file requests are sent or received in a single message. Bulk requests allow the system to guarantee a shorter number of message exchanges. In addition, the usage of HTTP allows further improvements in the payload processing by using HTTP streaming, similarly to the discussion on the dataset catalogue interface in Section 5.2.2.

In addition, the dataset master potentially holds a very large number of entries. Nonetheless, its index size is reduced because it contains only the requests waiting or in processing. Also, the dataset master implements a near online algorithm for the data flow decision that relies on relational database views. This allows for faster response to the data transfer requests. Finally, the system can be partitioned vertically, with the deployment of multiple dataset masters and a single redirection service. These implementation decisions help reduce the load per instance[19].

5.5 Administrative Services

This section briefly describes implementation details for the administrative services. The accounting service is not discussed since the current implementation consists of regular queries to the dataset master databases to generate usage reports regularly.

[19] In practice, the implementation described in Chapter 6 has shown to operate at rates higher than 50 Hz and the plots in Chapter 6 show a peak of 10 Hz of file transfers (with each individual file transfer consisting of multiple lookup and transfer requests between the dataset master and storage services).

Message	Description
File Transferring Message	Message notifying that a file transfer has been sent to the transfer tool in the storage services.
File Transfer Error Message	Message notifying that a file transfer has returned from the transfer tool with an error.
File Bad Message	Message notifying that a failed file transfer appears to contain a permanent error.
File Done Message	Message notifying that a file transfer completed successfully.
Service Error Message	Message notifying that a specific service has reported a failure.

Table 5.1: Monitoring messages generated by the storage services.

Message	Description
Subscription Complete Message	Message notifying that a dataset is now complete at a storage.
Subscription Queued Message	Message notifying that the transfer of a dataset has began.

Table 5.2: Monitoring messages generated by the dataset masters.

5.5.1 Monitoring Service

The monitoring service receives events from various components in the system. The main producers of events are the storage services. These events are sent using either HTTP or a messaging service. HTTP *callbacks* are based on GET and POST requests whose payload describes the monitored event. The messaging service sends the same event payload but uses the Java Message Service[20]. The centralised monitoring system is the consumer of events. These events are aggregated and displayed using a Web Interface. The resulting web interface is shown in the plots throughout Chapter 6. The list of monitored events sent by the storage services is shown in Table 5.1 and the events sent by the dataset masters are shown in Table 5.2.

5.5.2 Fabric Information Service

The fabric information service is responsible for storing information about the distributed computing fabric. Its schema is shown in Figure 5.15. As discussed in Section 5.3, the schema includes the `Old Address` and new `Address` for a `Dataset Master`, which is used whenever a redeployment occurs. The schema also contains information about storages. Each `Storage` is identified by a `Storage Name` and has a set of attributes (e.g. `Administrator Email`). A `Storage` may also have multiple `Alias` to aid in human identification. In addition, storages can be grouped into a `Cloud`. A cloud is a set of storages that can be referenced by a `Cloud Name`. These groupings assist in

[20] Refer to *http://java.sun.com/products/jms/*.

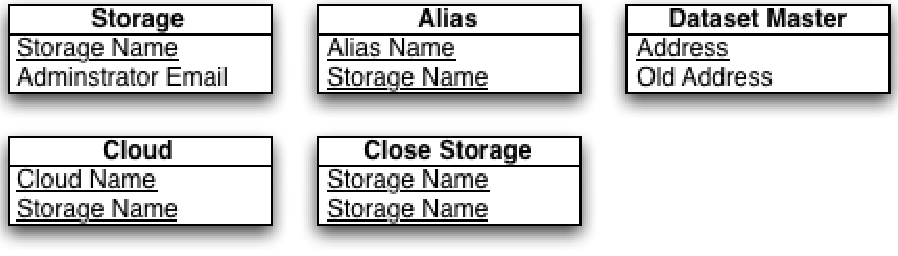


FIGURE 5.15: Schema of the fabric information service.

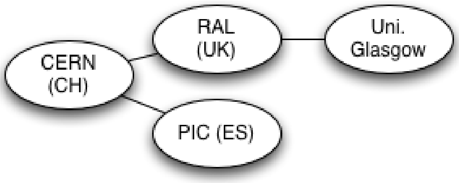


FIGURE 5.16: Example of close storages.

multiple operations. For instance, if the set of sources to use for a subscription forms a cloud, the user can use the `Cloud Name` directly in the subscription options, as opposed to the list of storage names.

Another entity in the Fabric Information Service is the `Close Storage`. This is better illustrated in Figure 5.16. Using graph theory terminology, close storages are the adjacent nodes in a graph, which are the nodes connected by an edge. In the figure, CERN is adjacent (or close) to RAL but not to the University of Glasgow.

Although logically part of the Fabric Information Service but implemented separately, there is a service that maintains the list of users and groups in the system. This uses the VOMS service [4]. VOMS is a Grid Service for managing authorisation data within multi-institutional collaborations. VOMS provides a database of user roles and capabilities and a set of tools for accessing and manipulating the database and using the database contents to generate Grid credentials for users when needed. Using VOMS, the proxy certificates used to contact to the HTTP servers in the dataset catalogue and dataset master are augmented with additional information, which includes the groups each user is assigned to. Groups have not been discussed in this chapter but can be used as part of the rules for the dataset master redirection services or the dataset catalogue naming services.

Various components in the system need to contact the fabric information service to obtain fabric information. (Several examples are shown in the pseudo-code for the storage services.) In the current implementation, whenever a client requests any information from the fabric information service, the entire information set is downloaded. This information is cached on the client-side and refreshed every few hours or on explicit request, as shown earlier in the pseudo-code. The current implementation is sufficient given the low volume of information contained in the information service. A disadvantage with the current implementation is that if a new storage service instance is deployed, there is a propagation delay until all services obtain this information. Nonetheless, this delay is similar to the propagation delay in the DNS service [97]. Alternative implementations could be based on the Globus MDS service [51] or in the more recent NodeWiz peer-to-peer resource discovery service [18].

5.6 Client Tools

The previous sections discussed the mechanisms to transfer and delete data in the system. However, the mechanism by which data is uploaded to the system for the first time has not been presented. The client tools are the entry point into the system. These are designed to be lightweight and with reduced number of external dependencies. Client tools provide a unified interface between the various components in the system using the interfaces for the dataset catalogue described in Section 5.2.2 and for the dataset master described in Section 5.4.2. Client tools allow users to upload data to the system and to download data out of the system to a local disk.

The process of creating a new dataset consists of importing files into a storage and creating the (logical) dataset definition. Besides choosing a dataset name, users must also choose logical file names and generate GUIDs for each new file. The client tools bring together these functions into a single unified interface: users are able to transfer files from their local disk to a storage and add these files to a dataset in a single operation. Internally, the client tools perform multiple operations: contact the dataset catalogue, the dataset master and the storage.

Similarly, client tools provide the ability to read out files from a storage system. Given a dataset name, users may download parts or the entire dataset to a local disk that is outside the managed fabric. This functionality is also provided under a unified interface: users provide a dataset name, and the client tools find a close storage with a dataset replica and download the files to the local disk. Both the import and export of files from a storage use the same transfer protocols and tools as the storage services (e.g. GridFTP). The difference is that transfers are no longer third-party because the data is uploaded or downloaded locally.

Another responsibility of the client tools is to provide the link between the dataset catalogue and the dataset master. Adding or removing files from a dataset involves only the dataset catalogue. But for this update to be applied to the physical replicas of the dataset, the dataset master must be contacted. This link is done by the client tools, which perform an extra call to the dataset master after every dataset catalogue operation. This ensures that updates to logical dataset definitions are reflected in the physical dataset replicas. Because the requests are *idempotent*[21], a failure in contacting one of the services can be retried safely. For instance, in the case of failure, it is possible that physical dataset replicas do not reflect a recent update to the logical dataset definition. Nonetheless, the event does not cause any data loss and can be safely retried later leading to the same result.

5.7 Summary

This chapter describes the implementation of the various components in the system. These are the dataset catalogue, the storage services, the dataset master, various administrative services and the client tools. The next chapter describes how this system is used in a production infrastructure for the ATLAS Experiment.

[21] That is, multiple applications of the same operation do not change the final result.

Chapter 6

System Evaluation and Simulation

"By a small sample we may judge of the whole piece."
(Miguel de Cervantes in *El ingenioso hidalgo don Quijote de la Mancha*)

In previous chapters I introduced an architecture and described the implementation of a system to manage distributed data for data-intensive applications. This chapter presents results of the real-world usage of this system, called DQ2[1]. DQ2 has been used to manage all the ATLAS Experiment data since 2005.

6.1 Methodology

A technique used to evaluate any system is to create conditions for reproducible experiments so that improvements can be tested in isolation of other system changes. In addition, the results obtained are usually compared to those of similar systems, to demonstrate where design differences have led to improvements or degradation.

The distributed, large scale nature of the work addressed in this thesis makes both conditions very difficult to implement in practice. It is not possible to have reproducible experiments if one intends to use the entire large scale infrastructure, since production-quality services must be maintained for the ATLAS physicists for their every day activities.

To circumvent this problem, computer scientists often deploy smaller scale infrastructures, which are separate from the real production infrastructure and contain only a

[1] DQ2 was originally designed with what was then "Windmill", the internal name for the ATLAS Monte-Carlo Simulation production manager application. The project name of "Don Quijote" (abbreviated to DQ, which is now at version 2) was adopted as an ironic reference to the conflicts that the fictional Don Quijote experienced fighting windmills [54].

subset of the resources. Unfortunately, this approach tends to miss many of the important behaviours that occur in real-world usage, because the scale is smaller and more importantly, because there are fewer external, chaotic factors influencing the system. These test infrastructures have fewer users, are more isolated, have a smaller total load and are maintained differently from the real production services.

Throughout this analysis, a major argument is that the infrastructure behaviour is for the most part unpredictable, because it is influenced by factors outside of system's control. A small scale experiment would likely miss these effects because there are fewer external users in a closed environment.

In addition, it is not easy to compare this system with other equivalent systems. DQ2 is managing over 14 petabytes of data distributed across data centres worldwide. As far as the author knows, there are no other open access results on the behaviour of similar systems; comparing two such systems in a fair and optimised manner would also be very difficult in practice, considering the scale of the resources required for such an experiment.

Therefore, given these constraints, I have opted for the following evaluation methodology. In the next section, I introduce the usage of DQ2. This starts by describing how DQ2 is deployed and how the distributed infrastructure is structured, followed by results obtained during real-world operations. The goal in this section is to illustrate the scalability properties of the system, by showing how the system has grown and how its overall performance has been maintained. Clearly these are high-level results, which are not entirely reproducible and may be subjected to questioning, but nonetheless I believe they provide significant evidence of scalability and fault tolerance.

The next section focuses on a more detailed analysis of how the infrastructure behaves. This is based on an analysis conducted for a shorter period of time under which a significant amount of data was gathered and analysed, both from DQ2 and from logs provided by several data centres. Some important results are described, which substantiate my thesis that any attempts to predict transfer behaviour - and any systems that rely on the prediction of transfer behaviour on a large, distributed, shared infrastructure - are most likely to fail, because there are underlying factors critical to the peformance, which are outside the knowledge of the system.

Despite the inability to predict transfers accurately, some form of simulation work is clearly useful in limited circumstances, such as during the development phase of the software: e.g. to test new functionality before it is put into production. The final section of this chapter illustrates how a simulator can be built, which captures more accurately the behaviour of the system.

Note: DQ2 is the system that implements the distributed data management system described in previous chapters. Nonetheless, there are a few differences between the

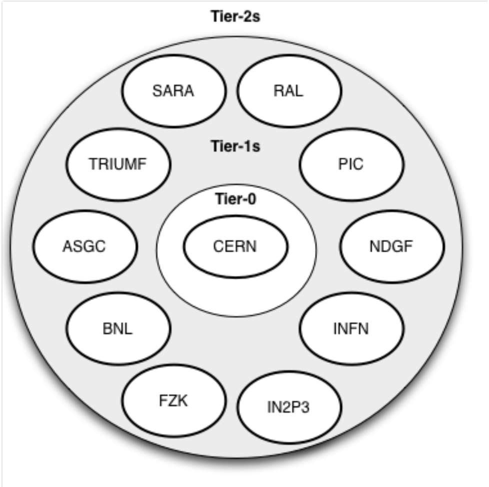


FIGURE 6.1: Representation of the Tier-0 and Tier-1s in DQ2.

architecture described in previous chapters and the system that is evaluated here. The system evaluated in this chapter corresponds to a slightly older version of the architecture. The newer version has not yet been put into production. The differences concern the deployment of the components: there is a single dataset master and dataset catalogue, which are hosted in the same database, and the partitioning of the namespace in the dataset catalogue is actually done at the database level and not by a separate service as described before. None of these differences significantly affects the scale at which the evaluated version has run, but the changes described in the architecture are expected to result in scalability gains by implementing additional partitioning of the components.

6.2 Usage

This section describes the usage of DQ2. It starts by describing how the system is deployed and how the underlying computing fabric is structured. This is followed by results obtained by using DQ2 to manage ATLAS experimental data, followed by a discussion on lessons learnt during this process.

6.2.1 Deployment

Figure 6.1 gives an overview of the data centre layers in the distributed computing infrastructure, which is called the Worldwide Large Hadron Collider (WLCG) Computing Grid[2]. As discussed in Chapter 3, the data centres in the WLCG are divided into multiple layers: the `Tier-0` centre, which is `CERN`, `Tier-1` centres and `Tier-2` centres (not shown). As shown in the figure, there are 10 Tier-1 centres for the ATLAS Experiment.

[2] Refer to *http://www.cern.ch/lcg.*

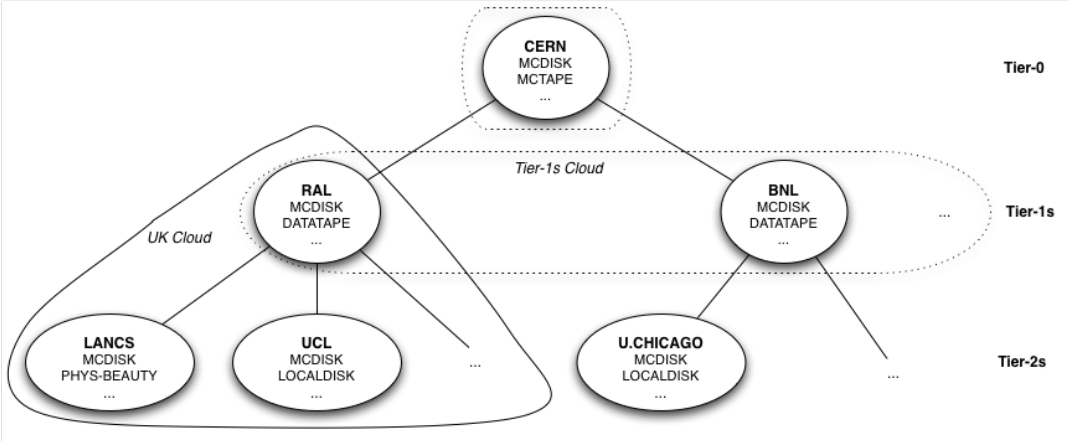


Figure 6.2: Representation of a subset of Tier-1s and Tier-2s in DQ2.

These centres are structured into clouds. Figure 6.2 shows only a small subset of the centres and their respective clouds. Clouds serve to implement the Tier organisation initially discussed in Chapter 3. For instance, the `UK Cloud` is constituted by the Tier-1 centre which is `RAL` (Rutherford Appleton Laboratory), plus several Tier-2 centres like `LANCS` and `UCL`: only a subset of the Tier-2s are shown in the figure. Another example of a cloud is the `Tier-1s Cloud` that groups all 10 Tier-1 centres.

The global services in DQ2, which are the dataset catalogue and master, are deployed at the Tier-0 centre, using an ORACLE relational database and Apache HTTP servers as described in Chapter 5. The storage services follow a slightly more complex deployment model, which varies from cloud to cloud. For instance, the `UK` Tier-1 and Tier-2 storage services are actually deployed at `CERN`, while the `US` Tier-1 and Tier-2 storage services are deployed at the respective data centres. The reason for this deployment model is strictly operational and follows from a negotiation involving operational team locations, shifts and execution of operational procedures.

These different deployment models are made possible in DQ2 because the storage services do not require privileged access to the storage. Instead, storage services make use of the SRM interface, as described in Section 5.3.1. Therefore, storage services can contact the storage from any location and establish secure (GSI) communication channels. All data centres that are part of the US cloud have opted for a different deployment model, which follows more naturally from the implementation described in Chapter 5. In this case, each data centre runs the storage services for its storage. Because the round-trip time between CERN and the US centres is higher, a local deployment seems more appropriate than that of the `UK` cloud, given the regular interactions between a storage service and its associated storage.

Figure 6.2 also shows additional names associated with each data centre. For instance, `LANCS` contains `MCDISK`, `PHYS-BEAUTY` among others. Each of these names corresponds to the actual storage services instance. That is, the storage at `LANCS` (Lancaster University)

Number of Tier-1s	10
Number of Tier-2s	82
Number of Storage Services	555
Number of Information System re-configurations	752

Table 6.1: Resources used by DQ2 as of June 10, 2009.

is internally divided into separate storage areas. Each of these storage areas is managed by separate storage service instances of DQ2. Each storage service instance has exclusive access to a subset of the storage, disk servers and namespace. The storage namespace is typically partitioned by having a unique base path such as `/atlas/phys-beauty/` for the `LANCS PHYS-BEAUTY` storage instance. Therefore, `LANCS` is in fact a cloud that aggregates multiple storage service instances.

The reason for splitting a storage into multiple areas is also strictly operational. Different working groups at a data centre often work with independent budgets and require exclusive access to their resources. But for data centre administrators, it is not desirable to run separate storage instances, so this partitioning is implemented instead at the DQ2 level. The disadvantage is that separating resources may cause unnecessary waste: a dataset may be present twice at Lancaster (e.g. requested by different groups and stored in their exclusive areas). In practice, this rarely happens because some of these storage areas host datasets that are likely to be requested by multiple groups and managers establish data flows in DQ2 so that these data are pre-placed automatically.

The network resources are not described in the figure but range from dedicated fibres to regular connections through the Internet. Between the Tier-0 and all Tier-1 centres there are dedicated network connections using an infrastructure called the LHC OPN[3], or the Large Hadron Collider Optical Private Network. The OPN is formed of dedicated 10 Gbit/s fibres. Between Tier-1 and associated Tier-2s, there is a mix of dedicated as well as regular Internet connections; between Tier-2s belonging to different Tier-1s, there are mostly regular Internet connections.

Table 6.1 describes the number of DQ2 storage service instances that have been deployed. Note that there are multiple storage service instances per storage (an average of about 6), as described in the Lancaster University example. These resources are known to the Fabric Information Service, which is also known as Tiers of ATLAS or ToA[4].

Since ToA was introduced in its present form in July 2006, its contents have been modified 752 times for re-configurations of the underlying infrastructure. This is a rather surprising number for a service containing information usually considered to be static, and implies that there have been configuration changes done on average every 34 hours. This illustrates the dynamism and growth sustained by the distributed computing

[3] Refer to *http://lhcopn.cern.ch*.

[4] The service was initially called ToA because at first it contained only tier and cloud associations for the ATLAS Experiment.

Number of Datasets	2,296,154
Number of Files	109,138,163
Total Data Volume	14,935.80 TBs

Table 6.2: Data stored in DQ2 as of June 10, 2009.

fabric. These modifications include the addition or removal of storage instances as well as modification of its properties, such as aliases, or cloud association.

6.2.2 Usage Results

This section describes results obtained by using DQ2 to manage the ATLAS experimental data. The first section starts with a presentation of overall usage results. This is followed by several sections, each focusing on a specific data management activity. For each of these sections the corresponding data flow is described along with performance results.

6.2.2.1 Overall Usage

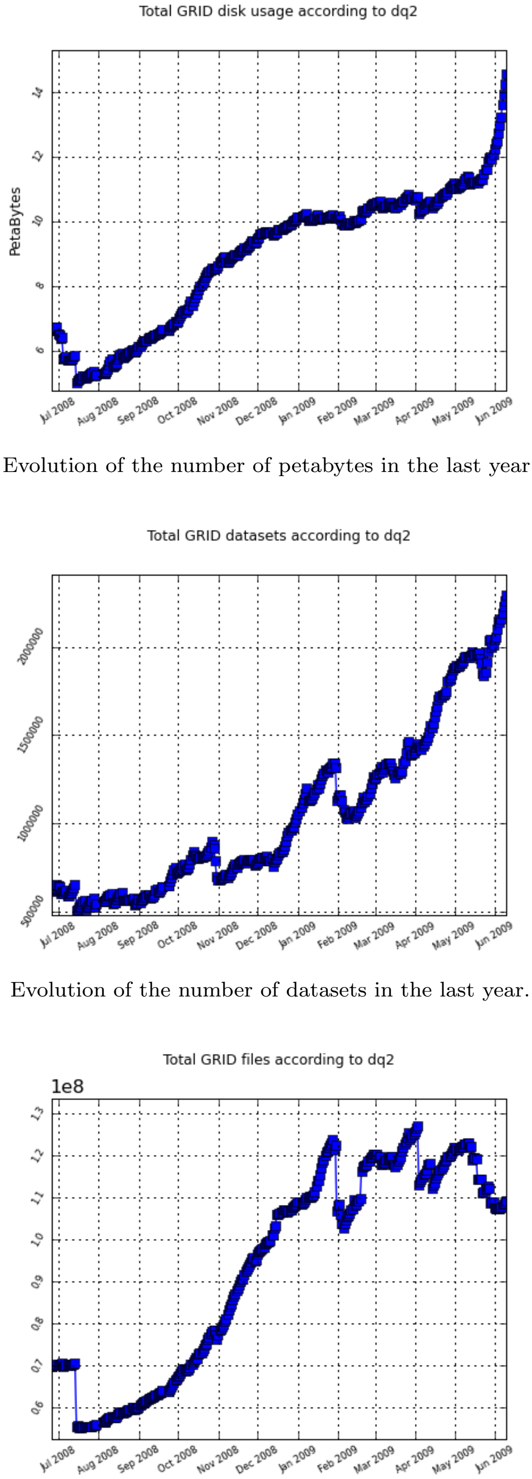
Table 6.2 shows the amount of data stored in DQ2. There are close to 15 petabytes of data stored in over 100 million files. The average dataset size is about 47 files. In practice, the distribution of datasets is more complex and some datasets are very large ($O(100,000)$ files).

The plots in Figure 6.3 illustrate the evolution of the data stored in the system. The stored data has grown steadily in the last year from 6 petabytes to about 15 petabytes. The only exceptional months are December and January where the production activities are halted or slowed down for the holiday period. The number of datasets has also grown from about 500,000 to over 2,000,000.

The plot in Figure 6.3(c) shows the variation in the number of stored files. The drops shown in the plot correspond to massive file merging activities that occur during reprocessing activities. In these activities, multiple data files are reprocessed but simultaneously merged together into a reduced number of output files, resulting in a drop in the total number of files in the system as the original inputs are deleted.

Figure 6.4 shows the increase in the data stored in DQ2 during a one month period. The sharp increase is due to large scale activities ongoing in June, as the final large scale tests are conducted before real data taking activities in summer 2009.

More importantly, the figures show the ability of the system to sustain growth, or scale, with the increase in the stored data. In particular, managing more data results in additional bookkeeping needs (dataset cataloguing, location, etc). As shown in the next



(a) Evolution of the number of petabytes in the last year.

(b) Evolution of the number of datasets in the last year.

(c) Evolution of the number of files in the last year.

FIGURE 6.3: Evolution plots of the data stored in DQ2 as of June 10, 2009 (plots provided by the Accounting Service and used with permission of Fernando Barreiro).

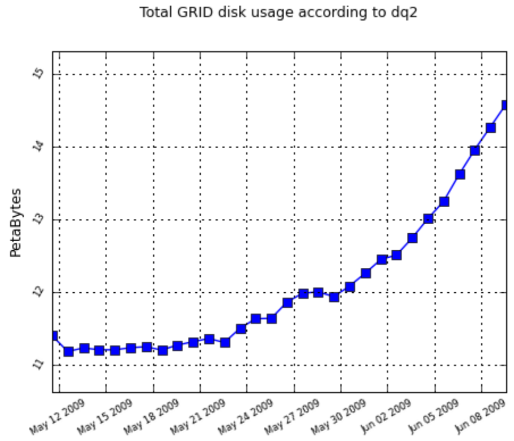


FIGURE 6.4: Evolution of the number of petabytes stored in DQ2 in the last 30 days, as of June 10, 2009 (plots also by Fernando Barreiro).

sections, these additional bookkeeping needs have not affected the performance of the system.

6.2.2.2 Data Export Activities

The data export activity is one of the main activities of the distributed data management system. It consists of distributing data from Tier-0 onto the Tier-1s and from there onto the Tier-2s. This activity occurs when the ATLAS detector is running and collecting new data (refer to Chapter 3 for additional information).

Before the LHC began to operate in the summer of 2009, there were a set of regular exercises conducted in the distributed computing fabric. These consisted of the generation of 'fake' detector data that is distributed exactly as real data. From the perspective of the distributed data management system, there is no distinction between distributing real or fake data, because the process is exactly the same, with the only difference being the physics contents (events) of the files.

This section shows the results of one of these large scale testing activities, called STEP09. STEP09 ran between June 2 and June 12, 2009. In STEP09, as with real data, new files (corresponding to new detector data) are written into the storage at CERN. These files are composed into datasets, where each dataset is formed by a set of files from the same physics process (e.g. same physics run, stream and "luminosity").

The data export activities follow a static dataset distribution pattern. Dataset transfer requests are inserted for all destination centres as soon as the dataset is first created at the Tier-0. Typically, each dataset is assigned to a subset of the Tier-1s and also to a subset of their Tier-2s, depending on the dataset type (e.g. RAW, ESD, AOD, etc). The data flows are pre-determined by the managers: the system cannot choose alternate sources for a transfer but must follow the assigned data flow. This includes the Tier-2s as well, which are assigned to receive data from its associated Tier-1 only. This mode of operation follows from a policy document called the ATLAS Computing Model and is discussed in Chapter 3.

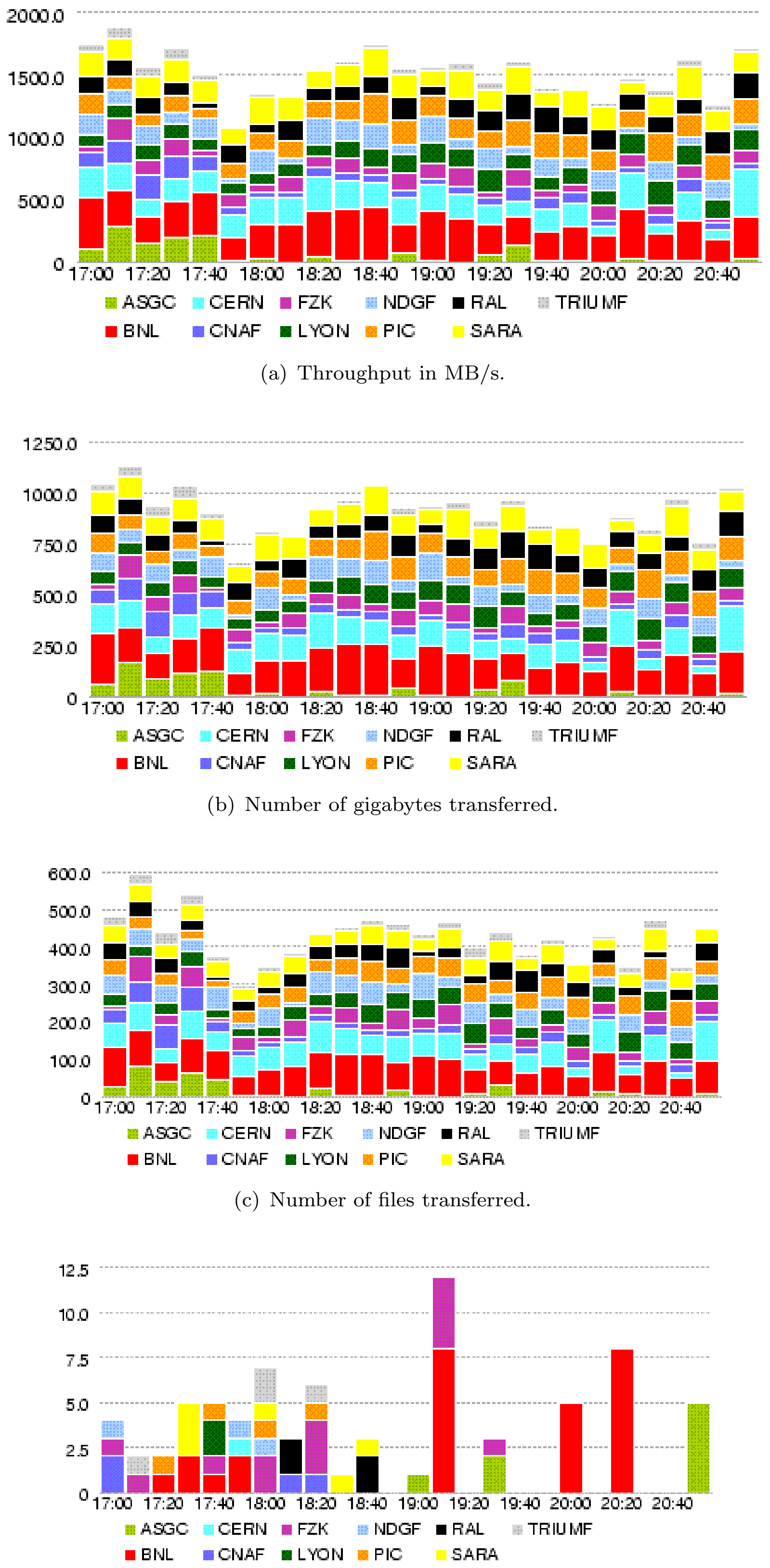
The distribution of data for the STEP09 exercise is primarily a throughput exercise because the distributed data management system must only ensure that data gets sent to the chosen destination as quickly as possible. It does not involve more complex dataset composition issues that occur in other activities, and which are described in the next sections.

Figure 6.5 shows some results of this data distribution. The plots describes a 4-hour period between 17h and 21h on June 10, 2009. The various Tier-1s are included in the plot. For each Tier-1, the values plotted correspond to the data each centre received from CERN. CERN is also included despite not being a Tier-1: this follows from a special role CERN has in these tests, functioning both as the centre of data generation (Tier-0) and also as an auxiliary Tier-1.

The throughput obtained in this period is shown in Figure 6.5(a). It reaches an average of approximately 1.25 GB/s. There is some fluctuation in the values, which can be attributed to multiple conditions such as instantaneous network performance, temporary degradation of disk servers or variation in the data acquisition process that produces fewer input data(sets). Nonetheless, close to 1 terabyte is transferred every 10 minutes, as shown in Figure 6.5(b). Also shown are the transfer failures in Figure 6.5(d). While these failures can temporarily degrade the performance, they are compensated by the transfer retry mechanism. The number of transferred files is shown in Figure 6.5(c); relating these values to the throughput shows that files are very large, typically several gigabytes each.

While the plots show good performance during a 4-hour period, it is important to verify whether this performance can be sustained over longer periods of time, where it is expected that storages will fail more regularly as the system is put into additional stress. Daily plots, from the 3rd to the 9th of June are shown in Figure 6.6. These include the throughput and the amount of data transferred. An immediate observation is that on June 8 the exercise was stopped for adjusting parameters related to the job processing mechanism, which are not related to the data management. Besides this event, the throughput is maintained, with about 100 terabytes of data copied every day.

To further analyse the scalability and performance of DQ2, it is important to check how additional flows impact the system. This demonstrates whether the system scales with additional requests that involve centres other than the Tier-0 or Tier-1s shown before. The plots shown so far correspond to transfer activities from the Tier-0 to Tier-1s only, but there are additional transfers that are part of the data export activities. One example that has been mentioned is the transfers from the Tier-1s down to the



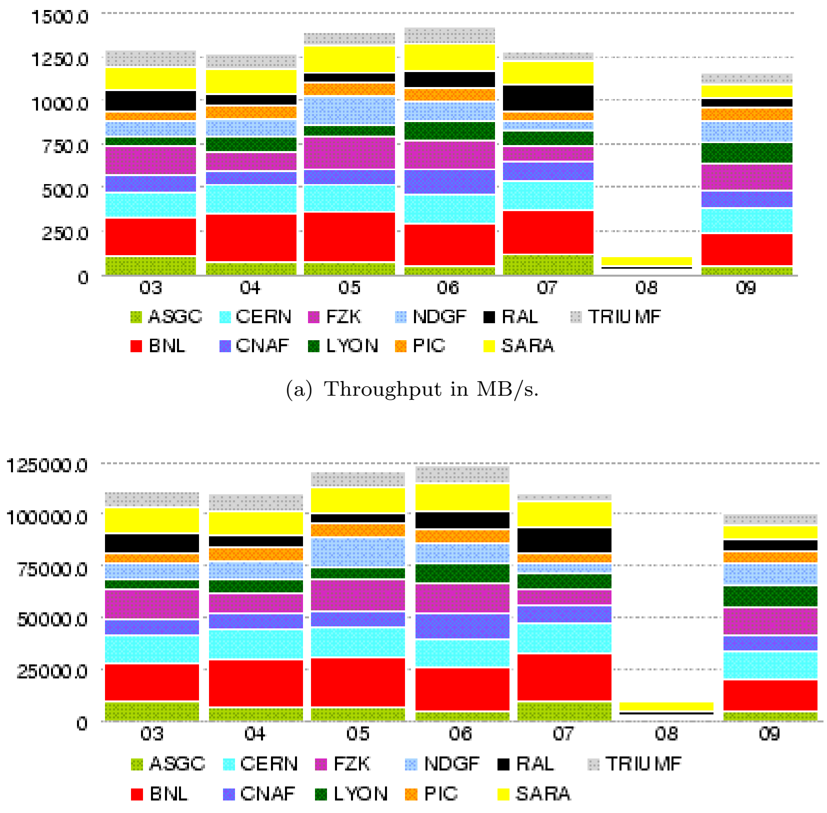
(a) Throughput in MB/s.

(b) Number of gigabytes transferred.

(c) Number of files transferred.

(d) Number of transfer errors.

FIGURE 6.5: DQ2 hourly performance during STEP09 (plots provided by the Monitoring Service and used with permission of Ricardo Rocha).



(a) Throughput in MB/s.

(b) Number of gigabytes transferred.

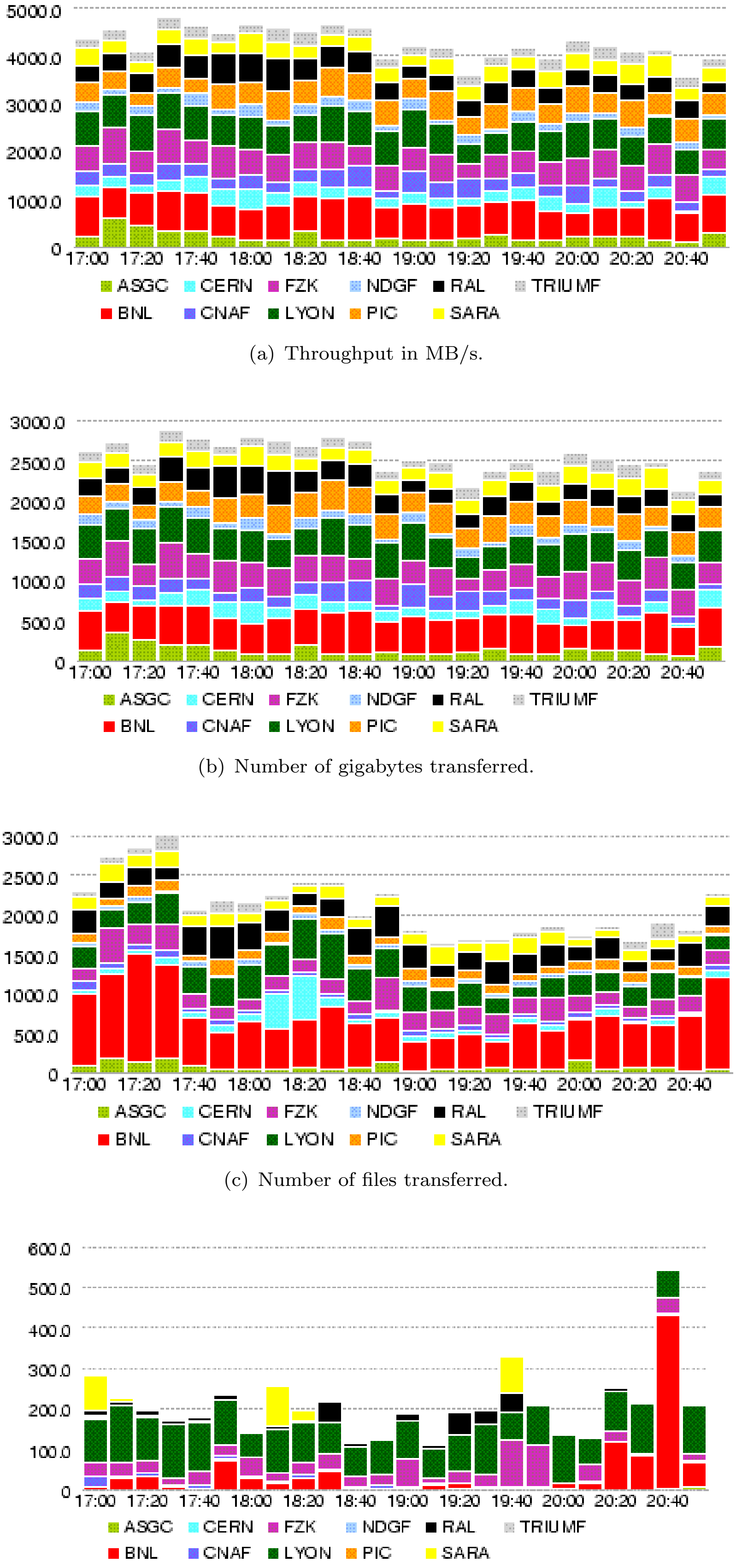
FIGURE 6.6: DQ2 daily performance during STEP09 (plots provided by the Monitoring Service and used with permission of Ricardo Rocha).

Tier-2 centres: when a Tier-1 receives datasets, some of these datasets[5] have to be sent also to the Tier-2s.

The set of plots in Figure 6.7 describes the same 4-hour period as in Figure 6.5 but accumulates all the different activities. The names ASGC, BNL, . . . , TRIUMF no longer correspond to the data centre name as before but to the entire cloud, which is formed by the Tier-1 and its associated Tier-2s. These plots show that for all data export activities, including Tier-0 to Tier-1s and Tier-1s to Tier-2s, the system achieves a sustained throughput of almost 4 GB/s, transferring close to 2.5 terabytes every 10 minutes. During these 4-hours, the system successfully recovers from transient failures, as shown in Figure 6.7(d). As before, except for small fluctuations, the rates achieved in these transfers are not limited by DQ2 but by the amount of requests available.

The final set of plots shows the time to complete dataset transfers. A desirable property is to complete the transfer of a single dataset as quickly as possible, so that processes that depend on this dataset can begin their processing tasks sooner. The scheduling and choice of data flows is discussed in Section 5.4.2. Results are shown in Figure 6.8. The majority of the dataset transfers complete in the first hours, either for a specific centre

[5] For instance the AOD datasets (see Chapter 3).



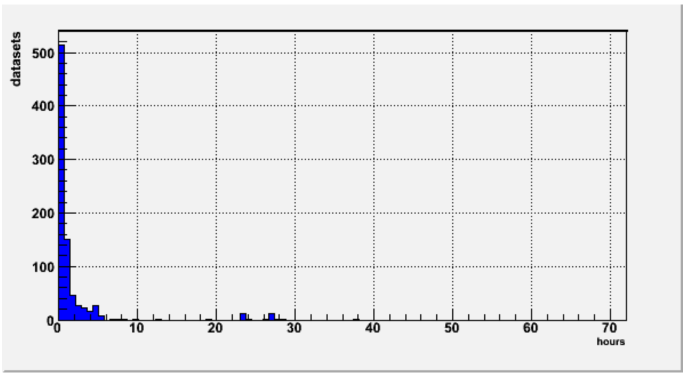
(a) Throughput in MB/s.

(b) Number of gigabytes transferred.

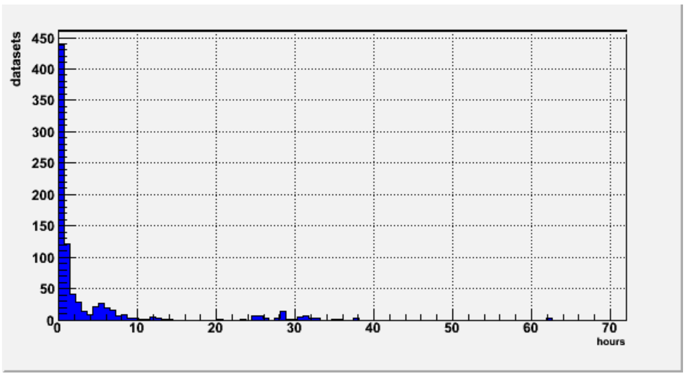
(c) Number of files transferred.

(d) Number of transfer errors.

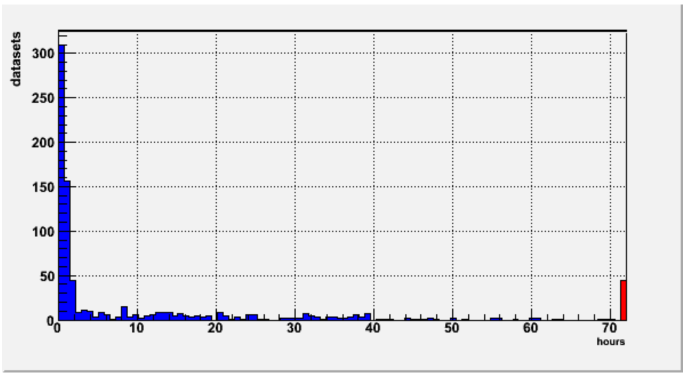
Figure 6.7: DQ2 hourly performance during STEP09 for all data management activities (plots provided by the Monitoring Service and used with permission of Ricardo Rocha).



(a) Number of hours to complete dataset transfers to the LYON Tier-1.



(b) Number of hours to complete dataset transfers to the SARA Tier-1.



(c) Number of hours to complete dataset transfers to the Italian cloud. (The red bar describes transfers incomplete after 70 hours.)

Figure 6.8: Time to complete dataset transfers during STEP09 (plots used with the permission of Alexei Klimentov and Alexey Anisyonkov).

(Figure 6.8(a) and Figure 6.8(b)) or for the centres in a cloud (Figure 6.8(c)).

The plots show that some transfers take several hours to conclude and some (in red) were not concluded when the plot was recorded. These are datasets affected by problematic files (permanent failures) usually caused by unavailable source files (e.g. missing or badly produced data). In these cases the dataset transfer will only conclude when these errors are fixed by operational times and the contents of the dataset changed.

The plots show that the majority of dataset subscriptions are completed in the first hour, but there is not a constant time to finish any dataset. Note that the production of datasets varies in time due to detector conditions and for STEP09 the datasets are produced in bulk. This causes a high load of requests and retries of failed transfers are intertwined with new requests, causing some delays.

6.2.2.3 Production Activities

This section describes another set of activities, which are the production activities, along with their usage of DQ2. The data export activities previously described are mostly concerned with high throughput data transfers. Production activities are instead characterised by lower data rates but more complex data management processes.

Production activities encompass all activities that involve the processing of data. For instance, Monte Carlo simulation (performed at Tier-2 centres) or data reprocessing (performed at Tier-1 centres) are examples of production activities. The data export activity previously mentioned is not strictly considered a production activity, because files are produced once (by the ATLAS detector machinery) but not using the distributed computing nodes (i.e. data is not produced on the Grid).

Figure 6.9 illustrates the workflow for production activities. There are multiple centres that generate or reprocess parts of the desired sample in parallel. In the figure, there are 2 `Tier-2` centres involved in the process. Each centre produces a distinct part of the sample. This is illustrated by each centre producing 2 files. These files are composed into a dataset at each centre; these are `Dataset A` and `Dataset B` in the figure.

The files are generated and added to datasets by the ATLAS production system, which is PanDA [125]. PanDA constantly sends Grid jobs to all ATLAS computing resources. As a job starts to execute in a worker node[6] (labeled as `CPU` in the figure), it contacts the PanDA server and requests work. This mode of operation is designated as *pilot jobs*. It allows for just-in-time scheduling as the job 'payload' (its task) is assigned at the last possible moment; in addition, by the time jobs start to execute and request payload,

[6] This description does not include the criteria used by PanDA to choose the data centres that will process specific datasets, as this decision relies primarily on processing and not data management requirements.

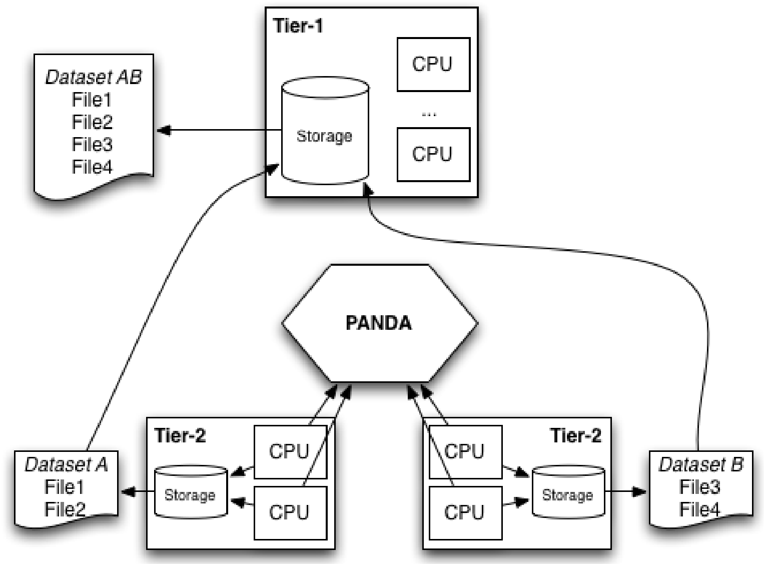


FIGURE 6.9: Illustration of workflow for production activities.

many failure scenarios have been surpassed since the job successfully started executing and is able to detect its exact operating environment.

Each job produces output files: in the figure, each job produces one output file per worker node; hence there are two output files in total for the two worker nodes. When jobs complete successfully, the results are reported to the PanDA server, which then adds each output file to the dataset. When the dataset is complete, the PanDA server requests the transfer of the dataset to its final destination. In this case, this will be the Tier-1 centre. At the Tier-1, a new dataset is formed that aggregates all the smaller datasets produced at the individual Tier-2s. This step is also coordinated by the PanDA server.

One feature of the distributed data management system is the ability to dispatch callbacks, or notifications, when certain events occur. In Section 5.5.1, these events were described in the context of the Monitoring Service. Nonetheless, other services can also request to receive these events, which are requested per dataset. The PanDA production system uses this feature to coordinate its production operations. When the input dataset required by several jobs arrives at a centre, PanDA is notified and releases the associated jobs for processing. These jobs are assigned to each worker node by PanDA after the pilot job contacts PanDA and requests the job payload. Later, when all jobs have executed and the dataset has been fully created at the Tier-2 (i.e. when all jobs for the sample are complete at a centre), the dataset is subscribed to the Tier-1. When it is fully copied to the Tier-1, PanDA is notified again by DQ2 and adds these contents to the new dataset. It may also request the deletion of the Tier-2 copy at this point.

The described workflow has shown to be robust with a separation of concerns between

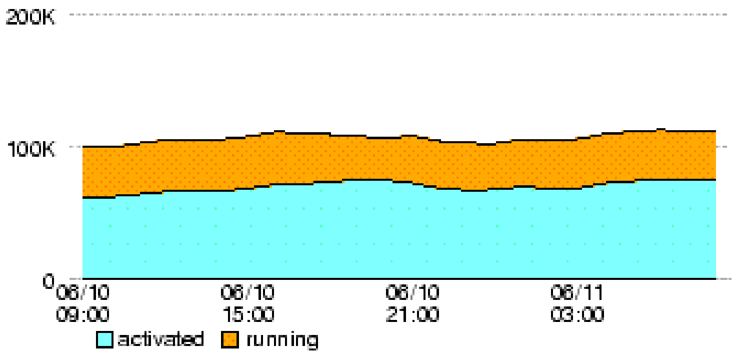
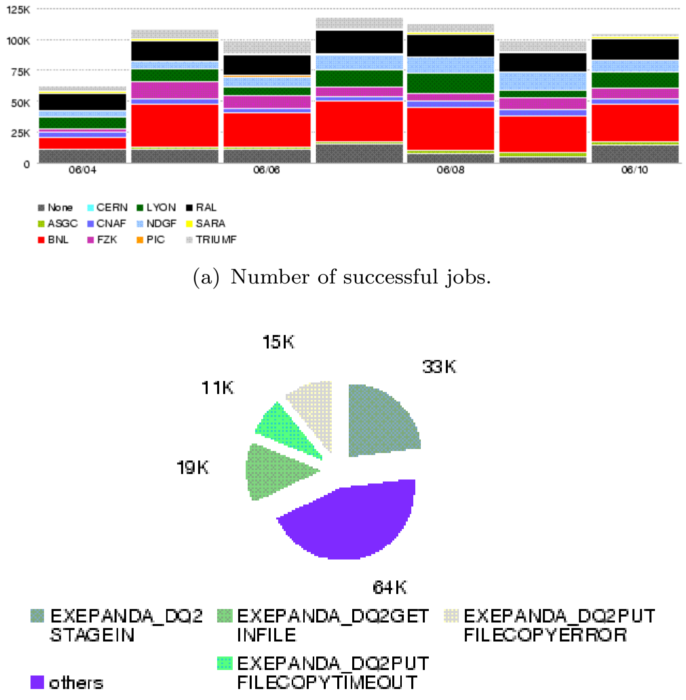


FIGURE 6.10: Queued jobs in last 24 hours as of June 11, 2009 (plots provided by the Monitoring Service and used with permission of Ricardo Rocha).

the distributed data management and production system, implemented by an interface based on notifications. Figure 6.10 shows some usage results. The plot contains a view of the running and activated jobs for all computing centres. Running jobs are the jobs in execution at a specific point in time. Activated jobs are jobs whose required datasets have been fully transferred and are now waiting for available CPU slots. As shown in the figure, there is a constant number of running jobs, which means the available computing resources are fully saturated. As such, it is possible to conclude that DQ2 is able to promptly transfer the required datasets: both entries in the plot correspond to datasets already available at a data centre, and the presence of activated jobs means there is a processing backlog.

Figure 6.11 presents a weekly view of the job execution. Each name in the plot (`ASGC`, `BNL`, . . . , `TRIUMF`) represents the cloud name, not the Tier-1 centre. More interesting is the plot in Figure 6.11(b) that shows failure conditions during one week. Many of these failure conditions are attributed to data management errors, such as reading input files between the local storage and the worker node, or writing output files to the local storage. None of these errors, despite the DQ2 designation, are a *distributed* data management issue but are due to problems contacting the storage locally at each data centre. An interesting conclusion is that even with jobs restricted to reading and writing data only from their local storage (and not from other remote storages on the Grid), there is a substantial number of failures (when compared to the total number of successful jobs).

Finally, it is important to note that these processing and transfer activities have occurred in parallel with the STEP09 exercise shown in the previous section. The competition between these two activities is handled in DQ2 by using fair shares: each activity is assigned to a share that has a specific percentage of the available transfer channel slots, as described in Section 5.3.4.2.



(a) Number of successful jobs.

(b) Failure distribution.

Figure 6.11: Weekly view of the job execution as of June 11, 2009 (plots provided by the Monitoring Service and used with permission of Ricardo Rocha).

6.2.3 Discussion

The previous sections discussed the usage of DQ2 for the ATLAS Experiment. One observation is that the infrastructure reveals very dynamic behaviour, with storages being added, removed and fairly static attributes changing regularly. This is illustrated by the number of changes to the information service.

Important observations from these results also concern the scalability of DQ2. The results demonstrate the ability of DQ2 to scale with the amount of stored data. This is illustrated with the STEP09 data export results. In addition, the results also demonstrate scalability with additional user requests. This is also observed from the STEP09 data export, which occurred in parallel to other background production activities. Finally, the ability of the system to support more complex flows is demonstrated by its integration with the production activities. In all these scenarios the system was also able to serve all the requested data promptly.

Despite these very positive results, it is not obvious to identify the parameters that most critically define the behaviour of the system, and in particular the performance of file transfers. The identification of these parameters is important to understand future bottlenecks. A similar observation concerns the analysis of the throughput. While the system demonstrates scalability and sustains current load needs, it is not clear if the performance obtained is adequate when compared to the available resources. For

Start Period	April 28th 2008
End Period	May 26th 2008
Number of Data centres	11
Successful Transfers	508,721 (96.7%)
Failed Transfers	17,424 (3.3%)
Number of Transfer Logs	526,145

Table 6.3: Overview of the analysis period.

instance, the Tier-0 to Tier-1 connections use dedicated 10 GBit/s network links from the LHC OPN, but the observed throughput between data centres is below this threshold. Therefore, it is reasonable to conclude that the throughput bottlenecks occur at the storages rather than at the network link. In fact, storage disk servers have a write speed in the order of tens of MB/s, e.g. 80-100 MB/s are common values. As such, the number of disk servers available in the import and export buffers impose an upper limit to the overall throughput rate. Nonetheless, there may be other factors such as the number of transfer slots that are allocated for each channel, or the distribution of these transfer slots per disk server by the SRM.

Based on these open questions, it is important to conduct a detailed analysis of the underlying fabric infrastructure. From this analysis I expect to identify the current bottlenecks at the fabric level, and understand the observed behaviour in greater detail. This is the subject of the next section.

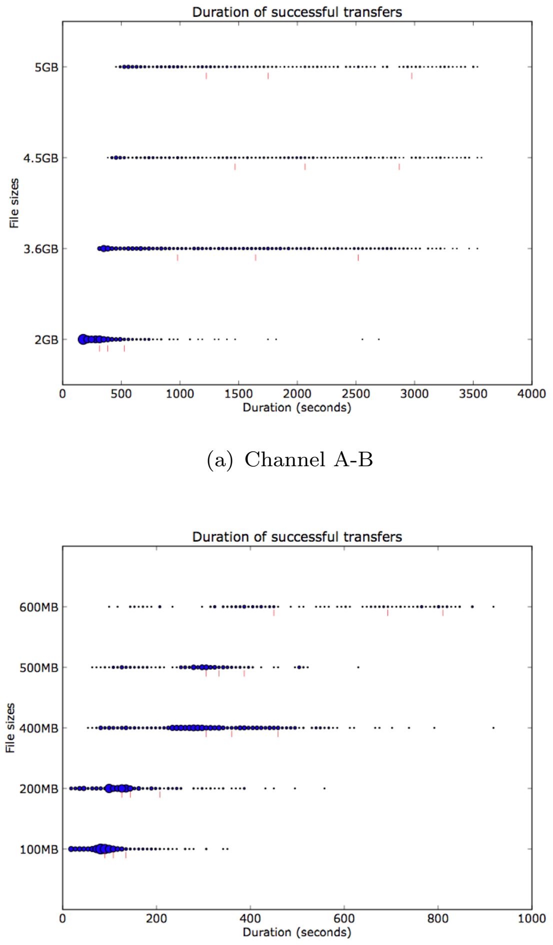
6.3 Infrastructure Analysis

In this section, I present an analysis of the behaviour of file transfers in the system. The objective is to understand some fluctuations in the transfer rates showed in previous sections. I start by describing the experimental setup, followed by an analysis of successful and failed transfers.

6.3.1 Experimental Setup

The results presented in the next sections correspond to the analysis of approximately one month of data, from April 29th 2008 to May 26th 2008. Only transfers between the Tier-0 and Tier-1 centres are analysed. These include Tier-0 to Tier-1, Tier-1 to Tier-0 and Tier-1 to Tier-1 transfers.

During this period, transfer logs from both DQ2 and the data centres were collected. The transfer logs of DQ2 include for each transfer, the request time, completion time, final transfer status and file information such as source and destination paths and file size. The data centre logs are the GridFTP and FTS server logs, which contain more detailed



(a) Channel A-B

(b) Channel C-D

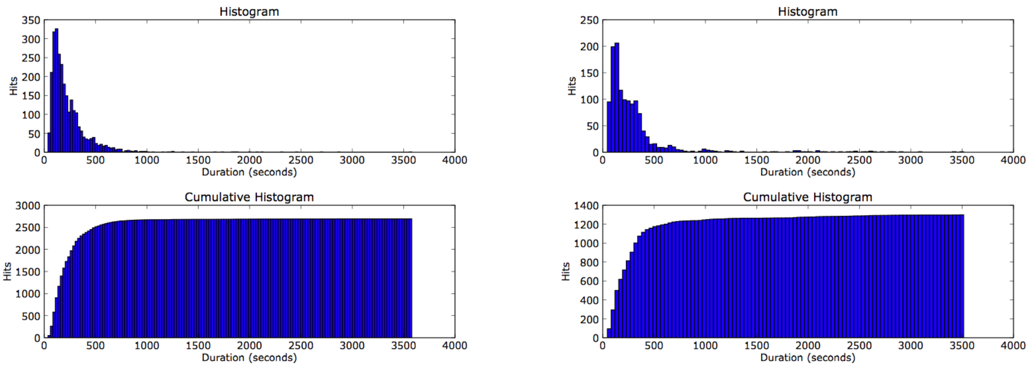
Figure 6.12: Scatter plot for duration of successful transfers for two different channels.

information such as the time taken to contact SRM servers or time spent in GridFTP for the network transfers. These logs were imported into a relational database and its entries correlated. Therefore, for each DQ2 transfer request the record also includes the GridFTP and FTS transfer information. Table 6.3 summarises the information collected.

For the remaining discussion and by request of some data centres, all results have been made anonymous. The channel A-B represents transfers from site A to site B.

6.3.2 Successful Transfers

Figure 6.12 shows the duration of successful transfers split by file size for two distinct channels. Two distinct channels with different file sizes are shown. There are immediate conclusions from these plots: transfer rates (here represented in seconds) vary greatly and the variations are sometimes counter-intuitive: e.g. bigger files may sometimes be almost as fast as smaller files; some of these variations may be due to statistical effects



(a) Channel E-F, transfers of 2 GB files

(b) Channel G-H, transfers of 2 GB files

FIGURE 6.13: Histogram for duration of successful transfers for two different channels.

but the overall trend remains.

For each file size, there are 3 red vertical markers. These represent the time up until, respectively 50%, 70% and 90%, of the transfers complete. A conclusion from the plots is that even if the transfer timeouts were reduced, which have been set to an artificially large value of 3600 seconds (1 hour), there would still be significant variations in rates. That is, even tolerating an additional 10% failure rate by setting timeouts at the 90% marker, some transfers would be 2 to 3 times faster than others. In summary, transfer rates have a significant tail.

Figure 6.13 shows the same situation (duration of successful transfers) but now only for a specific file size (2 GB files). Both histogram and cumulative histogram views are presented. The bin size for the histogram is determined using the Freedman-Diaconis rule [78] but the cumulative histogram is also shown since it is less sensitive to variations of the bin size.

Note that not all files are exactly the same size. Some file sizes vary by less than a few bytes. This is due to how the data is generated since each data file in ATLAS has a specific number of physics events and each event has the same size. As such, only minor variations occur in the sizes (e.g. due to different file headers). Again, the histogram shows a significant variation: in the cumulative histogram for channel E-F, half the transfers occur in under about 250 seconds and the remaining over 250 seconds.

A first assumption had been that transfer rates varied slightly but within a small interval. The expectation is that the distribution would follow a normal distribution with a target rate and rare occurrences in the tails. The plots for these 4 channels show that this is not the case: e.g. Figure 6.13(b) shows a more complex underlying distribution. The data gathered for several other channels shows the same pattern although the underlying distributions do not share the same parameters.

Following discussions with data centre administrators, we formulated the hypothesis that each disk server involved in the transfer was being simultaneously used for multiple activities: multiple parallel reads and/or writes by other users. These parameters are configured by administrators, often ad-hoc, with the goal of optimising overall storage usage and not specific patterns for specific users.

There are also multiple users (within ATLAS or even from other organisations) of the storage system and their joint usage causes a non-trivial interference on the disk server load patterns. To study whether this was the case, the following analysis was conducted. The objective is to observe how the transfer rates varied, for a specific file size, if the destination disk server was used in parallel for more than one transfer. From the DQ2 logs, it is possible to know *a posterior*, which disk servers were involved in transferring each file for a channel.

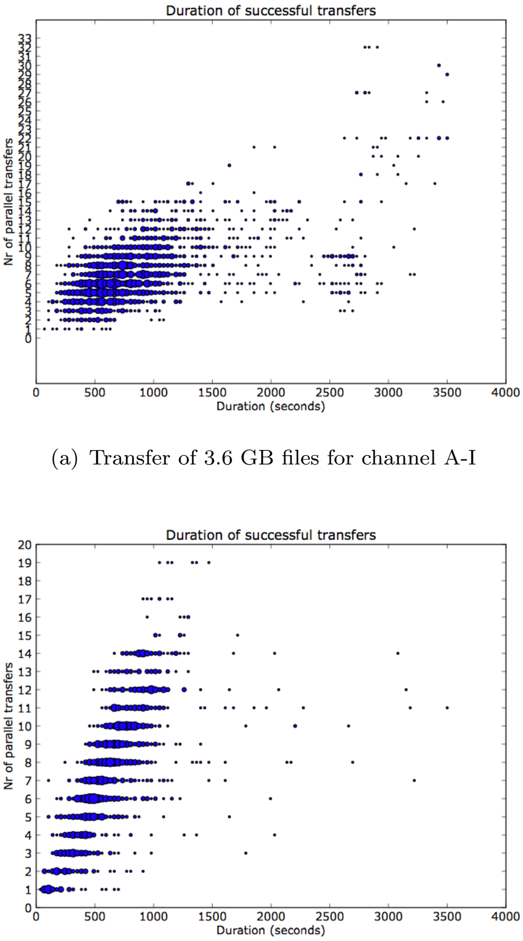
The expectation is that the transfer rates would decrease with parallel usage. In addition, it was expected that writes would cause more interference than read operations. As mentioned before, DQ2 does multiple parallel file transfers between data centres to compensate for underlying network limitations, such as the network round-trip time. Therefore, it is reasonable to assume that disk servers are busy with more than one transfer in parallel most of the time.

For this test, I only considered periods of time where ATLAS was the single or the major user of the storage system. Because this information is not known in advance, data centres were manually contacted and asked to report during which periods in the previous days ATLAS had been the sole user.

This method reduces the interference in the test, as only parallel disk server accesses done from DQ2 (i.e. from ATLAS) are taken into account. Figure 6.14 shows the results obtained for two channels. Clearly, as the number of parallel write operations increase on the destination disk server, the transfer rate decreases (the duration increases) and the variation is also wider. Note that destination sites have configured different sets of limits for parallel writes to a disk server: site I appears to have a maximum of at least 32 (as can be seen by the maximum value in the y-axis), while site J appears to have a maximum of 20 parallel writes. Based on this analysis, I concluded that to some extent, it is possible to understand after the fact the variations observed in transfer rates. In a next section I discuss whether it is possible or not to make use of this information but first I briefly analyse failures.

6.3.3 Failed Transfers

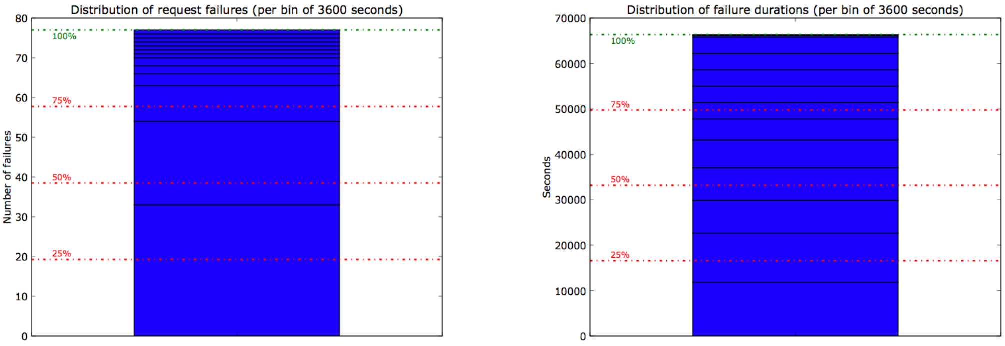
Regarding failures, the most important factors are how often these occur, the types of failures and most importantly, how many resources are consumed by the failure: that is, the duration of the failure. The failure duration is particularly important because while a failure occurs, no other file can be transferred since that transfer slot is busy.



(a) Transfer of 3.6 GB files for channel A-I

(b) Transfer of 3.4 GB files for channel A-J

FIGURE 6.14: Scatter plot for duration of successful transfers for two different channels.



(a) Number of failures

(b) Duration of failures

FIGURE 6.15: Stacked bins with failures for channel G-H

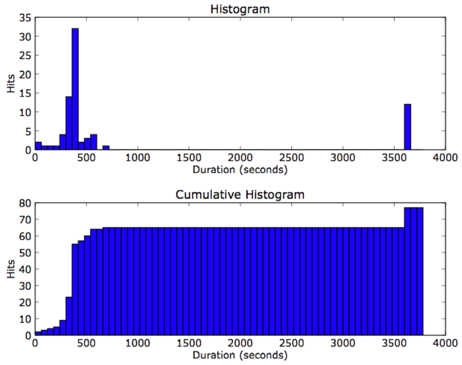


Figure 6.16: Duration of failures for channel G-H

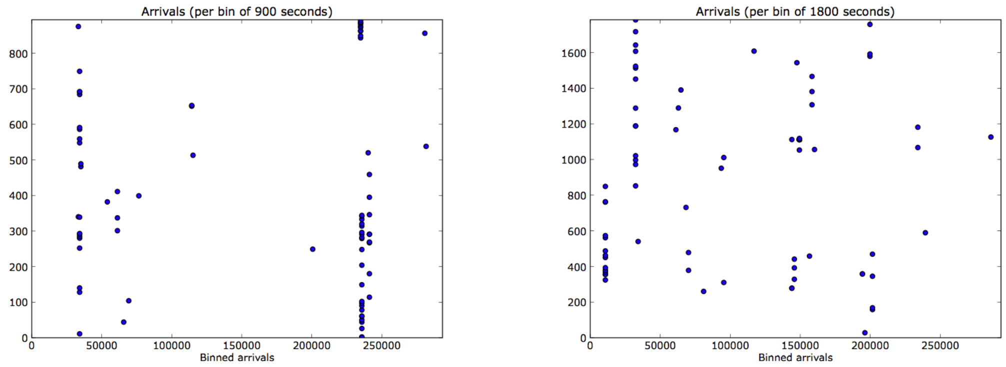
First, an analysis on the number and duration of failures is shown, followed by a discussion on failure types in a later section. Figure 6.15 shows a stacked plot with all failures that occurred in a 5-day period. Failures were added to bins and the figure shows these bins stacked and ordered by the large bins first. Therefore, the first bin in the figure, which is directly on top of the x-axis, corresponds to the peak of failures around the 300 seconds mark on Figure 6.16. What is evident is that a few bins contain most of the failures. In the figure, 3 bins cover over 75% of failures. This indicates that failures appear to occur in bursts.

The next bin on Figure 6.15 shows the same failures but now by duration. The dominance is less noticeable but still present. In the figure, 7 bins cover over 75% of the total duration failures. Also important is the amount of time spent in failures: a surprising total of 65000 seconds. This means that out of all the transfer slots used in a 5-day period, almost 18 hours of work were lost. Note that there are tens of transfer slots available in total. Nonetheless, the amount of time lost is non-negligible.

Figure 6.16 shows the corresponding failure histograms. Failures have variable duration, between 0 and 500 seconds and then there are timeouts, which were manually set to 3600 seconds.

An important factor is then to see how failures distribute over time. So far the analysis has shown that failures appear to have a burst behaviour. One of the difficulties in modelling failures arises from the fact that resources usually do not always fail completely[7]. Most often, resources degrade. That is, the services continue to function but e.g. one of the disk servers performs poorly (becomes slow or overloaded) but continues to serve transfers for some time before the storage system (or system administrators) detect and fix the problem. At the same time, all other disk servers in the storage continue to operate normally and hence, only a subset of the transfers that happen to use a specific disk server are affected.

[7] For this analysis, I removed periods of full downtime, since these were scheduled and announced in advance.



(a) Channel G-H

(b) Channel A-J

Figure 6.17: Arrivals of failures

Figure 6.17 shows the distribution of the arrival of failures for two channels. The x-axis is the (binned) time at which the failure occurred. The y-axis is the time, within the bin, at which the failure occurred. As an example: if a failure occurred at time 2500 using bins of size 1000, it would be plotted at coordinates $(x, y) = (\lfloor 2500/1000 \rfloor * 1000, 2500 \mod 1000) = (2000, 500)$. These plots are interesting because they show that failures do not appear to follow a Poisson distribution: if it were the case the plots would have a more or less uniform gradient, which is not the case.

6.4 Modelling and Simulation

The previous sections presented usage results of DQ2 and a detailed analysis of the behaviour of the infrastructure, through the study of file transfer performance. These results suggest the importance of having realistic models of the infrastructure during the development of the distributed data management system. Without such models, any developments that are perceived as improvements may not be adequate under real operational conditions.

Testing infrastructures are important to support the continuous development of any software product. Production use of DQ2 limits the ability to test new, potentially disruptive features. Similar limitations occur in other large scale systems. To address this issue, I developed an approximate model of the infrastructure and a discrete event simulator that uses this model. The objective is to allow DQ2 to be tested in a more realistic environment prior to production deployment, but also to provide modelling principles that can be used by the community at large. These modelling principles and the simulator are described in the following sections.

Various systems have been proposed to simulate distributed systems and in particular Data Grids. OptorSim [20] is an example of a Data Grid simulator built to study access to data from Grid jobs, in particular to help devise economic models on best replica placement strategies conditioned by limited storage space. Other examples of generic Grid simulators are Simgrid [38] and GridSim [163]. These simulators allow developers to build advanced analytical models for a Data Grid, by defining and parameterizing the behaviour of each resource in the Data Grid.

For instance, in GridSim [163] developers can configure in detail the performance parameters of every hard-disk and tape storage in the Data Grid. The work described in this section is complementary to these contributions in that it defines an alternative model that can be used by any simulator to model, in a realistic manner, the behaviour of transfers in a Data Grid.

The motivation for developing new simulation models results from the difficulty in applying existing models. In some cases, existing models require the precise definition of all resources in the Grid along its associated parameters (such as in GridSim). This is difficult to achieve for a very large, dynamic Grid infrastructure. Others (e.g. [21]) provide too simplistic network models, based for instance only in the network bandwidth. This is clearly not sufficient to characterise the observed behaviour. Some models do provide the ability to configure background traffic (e.g. [163]) but as described in this section, the observed behaviour is more complex and can be better characterised than background network traffic on a public network.

The main contribution in this section is the modelling techniques rather than the discrete event simulator. The simulator was mainly developed to help fine tune the model. In fact, it is in principle possible to apply the modelling principles to other simulators. On the other hand, the simple discrete event simulator does not support alternative models to be plugged in, which is a feature supported by most Grid simulators.

6.4.1 Modelling Principles

In a data management system, the hardest component to model is the underlying infrastructure, as is evident from the last sections. The goal is to model the behaviour of file transfers and how these are affected by the resources in the system. I restrict the resources to be the storage systems, network links and the transfer services in DQ2. The network links are represented by transfer channels, following the principles introduced in Section 5.4.2. This is a significant simplification of the overall infrastructure, which does not include for instance, the network routers or the individual disk servers in a storage.

In addition, I assume that any resource in the model can fail except for the transfer services. Therefore, failures can occur in the source or the destination storages or in the channel. I assume the transfer services do not fail but only the transfers themselves may fail. I believe this is a realistic assumption. Certainly for ATLAS, problems caused by the transfer services are extremely rare and usually due to hardware faults or temporary misconfigurations that are quickly detected. These events are not interesting for analysis.

Following the terminology introduced by [92], the model is divided into availability, reliability and performance modelling[8]. When performing a transfer, there are various possible outcomes:

- **The transfer is completed correctly.** In this case, the important parameter is the transfer duration. This corresponds to the *performance modelling*.
- **The transfer is refused.** For instance, the transfer does not start because one of the endpoints is not available. This corresponds to the *availability modelling*.
- **The transfer is completed incorrectly.** For instance, the transfer stops halfway leaving parts of the file transferred. Or the user abandons the transfer assuming it has timed out. This corresponds to *reliability modelling*.

6.4.2 Model for Successful Transfers

To model the performance of successful transfers I use a training set for each transfer channel. The objective is to derive a distribution from this training set, which is then used as a model.

Initially I assumed transfer rates followed a Gaussian distribution but the analysis conducted in previous sections demonstrated this is not the case. Nonetheless, discussions with system administrators highlighted the fact that the disk server load is a main factor affecting the variation in transfer rates. The tests previously described also showed this is likely to be the case, with different transfer rates observed with different numbers of parallel writes.

Disk server load is difficult to define and characterise in detail. It is not clear what other factors, besides the number of parallel writes, affect the load of a disk. The information available during my analysis did not appear to contain sufficient detail to identify additional factors. Nonetheless, while a gaussian distribution did not accurately represent the disk server behaviour, intuition indicated that a mixture of a reduced number of gaussians might be appropriate.

That is, as a result of discussions with system administrators, I formulated the hypothesis that disk servers can be represented as being in (coarse) load states. For instance, these load states could be cold, medium, hot disk servers. For each of these load states a gaussian distribution could be used to model the disk server behaviour.

[8] In [92], the term ‘speed’ is also used for ‘performance’.

Therefore, the following step is to create a Gaussian Mixture Model (GMM) [81]. A GMM is a probabilistic model for density distribution that uses a mixture of underlying distributions. In this case, the underlying distributions are gaussian. A GMM is appropriate in a situation in which there is the assumption of an underlying mechanism so that each observation belongs to one of some number of different sources or categories. In this case, the GMM defines a single distribution to represent the transfer performance, which itself is composed of several underlying gaussian distributions that intuitively correspond to each of the disk server load states.

The next step is to determine the parameters to use for the GMM. The Expectation-Maximisation (EM) algorithm in [61] is widely used for finding maximum likelihood estimates of parameters in probabilistic models, with frequent applications on data clustering and unsupervised learning. It is therefore an appropriate choice to determine the GMM parameters.

It is possible to generate multiple GMM models with a different number of underlying clusters. For instance, it is possible to generate a GMM with 2, 3 or more mixed gaussians. The EM algorithm can determine the maximum likelihood parameters for each of these classes, but cannot determine which is the "best" out of them. For instance, it is possible to determine the parameters for a GMM with 2 gaussians (i.e. 2 load states) using the EM algorithm, but it is not possible to readily conclude whether it is a better fit than a GMM that uses 3 mixed gaussians (i.e. 3 load states) instead.

Therefore, to deduce the appropriate number of clusters (of gaussians), I used the Bayesian Information Criterion (BIC) [156]. BIC is a criterion for model selection among a class of parametric models with different numbers of parameters. When estimating model parameters using maximum likelihood estimation, it is possible to increase the likelihood by adding additional parameters, which may result in over fitting. For instance, if in the example above the transfer channel were modelled with hundreds of gaussians, the resulting GMM would likely fit very adequately but it would result in a clear over fit that is too specific to a particular distribution. The BIC resolves this problem by introducing a penalty term for additional number of parameters in the model.

The results from the modelling of successful transfers are shown in Section 6.4.4. The next section discusses modelling of failed transfers.

6.4.3 Model for Failed Transfers

Failures are very difficult to model in a realistic manner. Unavailability (storage downtime) can be simulated although in practice these are often announced, scheduled and even negotiated. Unexpected downtimes have been extremely rare occurrences.

More common is the situation where the system is almost completely down but still some transfers succeed. In my model, I have considered this to be reliability, not availability modelling. Throughout this work I was unable to satisfactorily characterise failures and also distinguish between timeout failures (triggered by the user after some time) from other occurrences.

Another difficulty regarding failures is distinguishing between channel failures and storage failures. In practice, this distinction is difficult because the error messages are inconclusive. A channel failure should be normally due to a severe network fault that is not handled transparently by underlying protocols such as TCP.

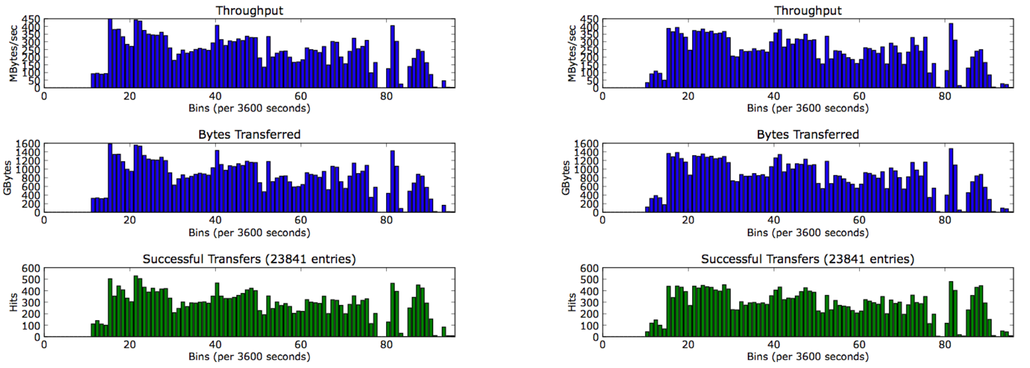
Therefore, I have limited the failure analysis to replaying failure conditions observed in the production system. This approach appears to give more realistic results than e.g. a Poisson-based model. Nonetheless, experience suggests that the best approach to validate data distribution strategies is to artificially create disruptive scenarios based on the perceived limitations of the strategies being developed. One useful technique has been to simulate a long downtime of a storage (a very rare occurrence in practice) to study the backlog recovery.

6.4.4 Simulator

This section describes the discrete event simulator. This simulator is developed in Python similarly to the rest of the DQ2 code. It implements an event scheduler and an event-driven approach, incrementing the time automatically to the next earliest occurring event [92]. (Appendix C contains relevant Python code excerpts from the simulator implementation.)

At start-up, the discrete event simulator reads historical transfer logs from real transfers. These logs are stored in a relational database and follow the same structure as the transfer logs described in Section 6.3. These logs are used as the training set to determine the model parameters for the GMM described in the previous sections. Therefore, at start-up the discrete event simulator determines the GMM distribution to use for each channel, based on the analysis of historical logs and the application of the EM algorithm and BIC criterion. It also determines the occurrence of failures, by analysing the mean time between failures from historical data.

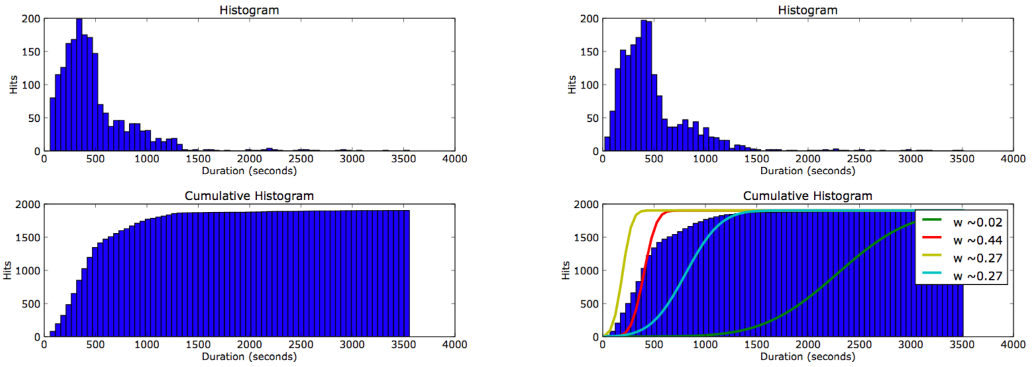
In addition to the training set, the discrete event simulator also reads at start-up its workload from a relational database. The workload consists of transfer requests that are to be received by the storage services at a specific point in time. These are processed as transfer requests by the discrete event simulator when its simulation clock reaches the transfer request time. At that point the discrete event simulator simulates the transfer outcome (success or failure) and duration of event based on the model parameters. This is reported back when the event is complete according to the simulation clock.



(a) Channel A-I: Observed results

(b) Channel A-I: Simulated results

FIGURE 6.18: Overview of both observed and simulated throughput results.



(a) Channel A-I: Observed distribution

(b) Channel A-I: Simulated distribution

FIGURE 6.19: Distribution of successful transfers, 2 GB files

In the remaining part of this section I validate the model used for the simulation. For this, the set of transfer logs analysed in Section 6.3 is re-used. These logs are partitioned into two separate sets. The total set of transfer logs constitute 1 month of transfers. The first week of data is used exclusively as the training set, and the remaining 3 weeks are used as the validation set.

The plots in the following figures include only the comparison between observed and simulated events for 4 days out of the 3 weeks of validation data. This is done for readability purposes, since plotting 4 days out of a period of 3 weeks allows the plots to have higher resolution: in this case, 1 hour bins.

Figure 6.18 shows an overview of the observed and simulated throughput. Because the mean time between failures was used, the exact occurrence of a specific failure can vary slightly, but the total number of failures remains the same (hence, so does the total number of successful transfers shown in the figure). The overview shows visually what appears to be a very good match, which is nonetheless influenced by the bin size chosen for the plot. Therefore, a more detailed analysis is required.

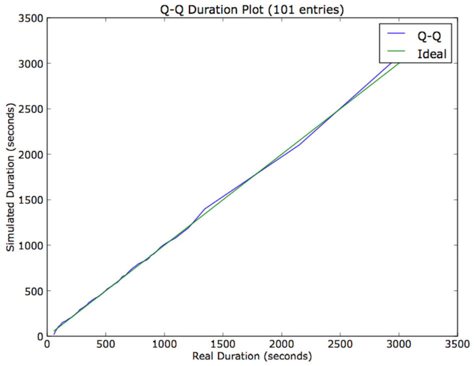


Figure 6.20: Quantile-Quantile plot for distributions on Figure 6.19

Figure 6.19 shows the observed and simulated distribution of successful transfers of 2 GB files. This figure allows for a more detailed analysis of the model. The gaussians that compose the simulated mixture model are also plotted and scaled for illustration purposes. Figure 6.20 shows the corresponding Quantile-Quantile (QQ) plot, comparing both distributions. In a QQ plot, a line overlapping with the ideal line represents a perfect fit between the distributions.

These plots show a very good matching between the observed and simulated data. The distributions are generated exclusively from the test set data, but the comparisons are performed using workload from the validation set. This indicates that the simulator appears to adequately model the real infrastructure.

Nonetheless, there are important limitations in the approach taken to build the model and simulator. The fact that there is a close match between observed and simulated distributions is due to the weights derived by EM algorithm and the clusters identified by the BIC criterion. The examples shown are for a single channel but I observe successful fits for other channels as well.

Nonetheless, the parameters for each GMM can be very different from channel to channel. I did not find any specific relation between channels. Nonetheless, the fact that the best fit is always achieved with a reduced number of gaussians (3 or 4) gives some degree of confidence in the initial intuition, which informally characterises the disk servers into separate load states.

Most importantly, the primary goal of this simulation work is to achieve a model closer to reality, which can be used to test DQ2 in a simulator. The goal is not to develop a detailed analytical model of a distributed infrastructure nor to use previous transfers for prediction. Nonetheless, some important lessons can be learnt from this analysis. These are discussed in the following section.

6.5 Distributed Data Management under Uncertainty

The previous sections illustrated the uncertainty present in the underlying infrastructure and presented an approximate model of the overall system. This section discusses whether it is possible to make use of this knowledge to improve the distributed data management system.

While it is possible *a posterior* to understand some of the observed behaviour, there is limited ability *a priori* to take any action. For example, when transferring a file between two storage systems, the allocation of the source and destination disk servers only occurs when the transfer starts. It is also not known if there are other users using the same disk servers. Therefore, there is no mechanism to control one of the main factors of the transfer performance, which is the load on the disk servers if we assume we are operating using distributed and shared resources.

One option could be to reduce the number of transfer slots. This solution is of limited interest: the network round trip time would affect the overall transfer performance enforcing low upper bounds on the throughput. In addition, it would still not be known if there are other users of the disk servers, including non-ATLAS users. It would also not be known how intensively these users are using the infrastructure. There is also no control on the internals of each storage system: note that storage systems broker requests to choose the most suitable disk server at a point in time using internal criteria. Finally, deploying fully separate infrastructures for different users is not a cost-effective option. Nonetheless, this principle is applied throughout many areas of the infrastructure (e.g. job queues, hierarchical storage management, etc).

In reality, the uncertainty is due to the nature of distributed and shared computing. Computing centres try to optimise their global performance while each organisation, with limited information, tries to optimise its own perceived performance.

Most of these considerations are only relevant if the goal is to have a system that aims to provide some guarantees of quality of service, such as DQ2. One of the most used features in DQ2 is the ability to allocate the competing transfer requests to channels fairly. If a physics discovery leads to an increased interest in a specific activity, the data(sets) from this activity should be distributed as fast as possible so that future processing activities can start. Nonetheless, as observed, failures are very unpredictable. A timeout may prevent a dataset from being completed because one of its files is not available due to a timeout error.

There is a conclusion that follows from this discussion. If there are factors that fundamentally affect transfer performance, which cannot be known in advance, then the DQ2 design decision of not scheduling based on transfer prediction is quite appropriate. DQ2 implements a feedback-based approach. That is, rather than trying to predict the transfer times and schedule based on predictions, DQ2 instead reacts to finished transfers. When a transfer is complete, storage services ask for additional work. Based on the analysis of the infrastructure this appears to be a more robust approach.

6.6 Summary

This chapter presented and analysed the performance of DQ2, which is the distributed data management system for the ATLAS Experiment. The deployment scenario and usage results were introduced. These results include overall performance figures as well as a detailed analysis of specific data management activities, which range from high throughput transfers (e.g. STEP09) to more complex data management workflows (e.g. production activities).

These overall usage results provide very positive evidence regarding the scalability and performance of DQ2. The amount of data stored in DQ2 is in the order of 14 petabytes with a transfer throughput of 4 GB/s. Nonetheless, the results do not contain sufficient information to deduce the factors that most influence overall system performance. Therefore, an extensive analysis of file transfer performance was conducted during a one-month period. These results are also presented in this chapter. There are several important conclusions from this analysis, which substantiate the premise that transfer performance is affected by factors outside the system's control, such as the number of parallel reads and writes in a disk server.

The final sections of this chapter investigate the use of modelling and simulation techniques. Such techniques are required to test new versions of DQ2 without disrupting the production facility. Existing Grid simulators either do not provide sufficiently realistic models (e.g. simplistic models that do not capture the observed file transfer behaviour), or are very difficult to apply in practice (e.g. require a large amount of low-level information on the distributed computing fabric). Therefore, I developed new modelling techniques for file transfer performance. These techniques are presented in this chapter. The resulting models (and corresponding simulator) include the performance, availability and reliability of file transfers on the wide-area network. The validation of the simulator with real transfers demonstrates that it adequately models the observed behaviour.

Finally, I discussed the problem of distributed data management under uncertainty. Throughout this chapter I identified several factors that influence the performance of the system. Some of these factors are outside the system's control. Nonetheless, these factors critically define its performance. Therefore, it is important to discuss whether the uncertainty in the infrastructure imposes practical limits to the development of a distributed data management system. I believe this to be the case. Finally, I established an important conclusion on the design of DQ2: if there are factors that fundamentally affect transfer performance, and which cannot be known in advance, then the DQ2 design decision of scheduling based on a feedback approach appears to be the more robust solution.

Chapter 7

Conclusion and Future Work

7.1 Contributions

I now review the main contributions of this work.

- Chapter 4 introduces a set of design principles that can be applied to the creation of a non-intrusive and scalable distributed data management system. The set of design principles are:
  - The introduction of datasets, which are loosely defined as collections of files, as the underlying data unit. Therefore, systems that employ proprietary data models can easily integrate with the distributed data management system without changing the existing data flows.
  - The adoption of (eventually) consistent principles for the replication of datasets across data centres.
  - The separation between logical and physical data units, allowing (logical) dataset definitions to change independently of physically stored data, and having dataset updates eventually propagate to the various physical replicas.
  - The definition of fabric independence, or the ability to change any part of the underlying distributed fabric (architecture of data centre, location of the data, etc) transparently to the users.
  - Following from the previous design principle, the definition of a layered system that does not require modifications to the middleware already employed at a data centre.
- Based on the set of design principles, Chapter 4 and Chapter 5 describe an architecture and implementation of a system that is scalable, fault tolerant and secure.

- Chapter 6 evaluates the system proposed, which is called DQ2. DQ2 is currently managing over 14 petabytes of data distributed across data centres worldwide. In addition, the real world usage of DQ2 for the ATLAS Experiment has shown that the system is able to scale with the amount of stored data, maintain adequate performance and is integrated with diverse data flows of various degrees of complexity.
- As a consequence of the real world usage of DQ2, Chapter 6 also proposes new principles to model the behaviour of transfers on a Data Grid. These algorithms are based on the generation of Gaussian Mixture Models, and employ statistical methods to model more accurately the behaviour observed. These models have the advantage of being simpler to apply, compared to existing simulation tools.

Finally, in this thesis I have made an attempt to describe more accurately the problem of managing distributed data. This is described throughout this work, from the collection of requirements to the identification of the uncertainty that underlies a shared Data Grid environment, and includes the identification of practical limits to the development of transfer algorithms.

7.2 Future Work

In this section, I identify two areas for future work. These areas are a logical continuation of the work presented in this thesis. The first area focuses on exploiting data access interfaces. The second area aims to improve the behaviour of the lower layers by introducing new mechanisms to route very large datasets. I discuss each of them in turn.

7.2.1 Data Access Interfaces

As data moves into "the cloud", the problem of managing distributed data will certainly grow. As argued in Section 2.4.4, cloud systems already include a growing convergence between file systems and database technologies. I believe developer APIs will become more relevant in the near future. These APIs must be simple to use, provide rich functionality but still allow cloud providers to independently store and transfer data across data centres or even across clouds.

POSIX-style access has brought distributed systems a long way, even with some success in niche use-cases when applied to the wide-area network. Nonetheless, it is of interest to re-appraise this approach. In this thesis, the introduction of datasets has enabled the implementation of a simple, distributed and easily partitioned system. Other approaches may be possible, which can also take advantage of both structured and unstructured data.

As such, an area of future work concerns the creation of new data model paradigms that allow for the network latency limits to be circumvented in the wide-area network. This may lead to the creation of new transaction models that exploit the internal organisation of the data, or perhaps the data processing primitives.

In addition, and expanding on the research initially presented in [28], these data model and data processing primitives could include transparent mechanisms to record provenance information. This provides a tighter coupling, and results in a more functional distributed data management system for storing structured information.

7.2.2 Routing of Very Large Datasets

In Section 1.3, I stated that an important component of this work is an analysis of the systems that rely on predictive frameworks as well as simplistic models. The difficulty is in creating (mathematical) models that accurately represent a large, dynamic and shared infrastructure. A parallel can be made to the domain of network routing, where various theoretical contributions have been developed.

Therefore, a line of future work is to adapt and expand the work that has been done within the network routing domain onto the *routing of very large datasets*. There are important differences between the two: network routing focuses on routing small packets, where routing decisions must be made very quickly and where routers have very limited buffer space. Routes also do not share the same access costs to the data (e.g. recalls from tape) as storage systems do. Nonetheless, important parallels can be established between the two. In network routing, a major challenge is in predicting and/or reacting to the workload. In addition, network routing aims to establish paths between source and destination, so that future requests follow the same paths. There are clear parallels to the routing of very large datasets. Additional parallels can made between routers with limited buffer space and storage systems with limited disk space. This line of work is more relevant if there are the goals of not assuming a completely connected network, as in DQ2, and if the choice of sources for replication is moved from being primarily (but not exclusively) on the user side, to being on the system side.

It is also instructive to discuss some important developments within the network routing domain. There are two fundamental approaches to routing in networks. One is to route data depending on the current load in the network (i.e. *adaptive routing*). The alternative is to route data independently of the current state of the network (i.e. *oblivious routing*). Adaptive protocols can achieve reduced network congestion, but are harder to implement in a distributed system.

In [11] the authors design a routing algorithm that is *log(n)* competitive with respect to congestion. It is an adaptive, centralised algorithm that serialises routing requests. [12] introduces a distributed algorithm that chooses routes by repeatedly scanning the network. The algorithm requires shared variables for each edge, and hence is considerably more difficult to implement.

In a more recent paper, Räcke [136] introduced the surprising result of a competitive oblivious routing algorithm for general undirected networks. This result means that it is possible to come close to minimal network congestion without any information on the current load in the network. This oblivious algorithm has been constructed in [14].

There is considerable potential in exploring Räcke's oblivious algorithm for higher-level applications and in particular the principle of hierarchical decomposition for congestion minimisation recently presented in [137]. When routing large datasets instead of network packets, the network congestion can instead be a function the number of transfer slots in a transfer channel. This results in a novel use of the transfer channel concept as a congestion model. In addition, the fact that the algorithm is oblivious to the workload addresses (partially) the problem of uncertainty discussed throughout this thesis.

Alternative approaches to the problem may derive from exploiting other forms of randomised algorithms, or adapting approaches to network routing that rely on game theoretical principles [131].

7.3 Concluding Remarks

A team of physicists and engineers in Geneva tune the data taking process of a complex detector. The data taking process incorporates real-time processing to filter out events of no significance, but early trials reveal that it is able to discard far too few events. As a result, the data taking volume is several times what was expected. Meanwhile, physicists in Orsay, Glasgow and Texas collaborate on the production of a large statistical sample that should show the detection of a new physical behaviour. The deadline is short, because the results are to be presented in an upcoming physics conference. At the same time, managerial meetings decide to invite a new computing centre to join the experiment, exchanging additional computing and storage resource by privileged access to experimental data.

The question addressed in this thesis is how to integrate these conflicting interests and building a unified data management system? Users need undisturbed access to the data; managers are concerned with overall data storage requirements, while data centres are always busy dealing with the needs of diverse communities of users.

In theory, the data management problem can be addressed by devising integrated data storage and processing systems. Experimental workflows and algorithms can be redesigned and adjusted to the underlying technical constraints. Computing resources can, in theory, be centralised into a single large computing centre, and the work carried out in a near optimal fashion. In practice, this solution is of little interest given the constraints set by policies, budgets and even geography.

In [84] Jim Gray et al argue that the file *modus operandi* will just not work at the peta-scale. They argue that "some new way of managing and accessing information is needed" and that "metadata is the key to this". While I disagree that the file *modus operandi* does not work at the peta-scale - the proposed system in this thesis does work at the peta-scale and is, at its lowest level, based on files - the key to my proposed solution has indeed been the opportunistic use of metadata in the form of datasets as collections of files. This thesis builds upon these ideas, and presents a set of basic design principles out of which a subtle exploitation of metadata is fundamental. These principles have been successfully applied in a straightforward implementation that has served the practical needs of a large running experiment. The resulting implementation has successfully managed tens of petabytes of data, in perhaps one of the largest data management systems built to date.

Appendix A

Dataset Catalogue API

The dataset catalogue interface includes the following methods:

- `create_dataset(dsn)`: Creates a new dataset definition, given a dataset name (`dsn`). The first version of the dataset is created (`version 1`) and the dataset state is set to `OPEN`. The dataset contains no files at this time.

- `add_files(dsn, list_of_files)`: Adds logical files to the dataset with name `dsn`. The dataset state must be `OPEN`. The `list_of_files` is a list of tuples with `(GUID, LFN)`. The GUID is the Globally[1] Unique Identifier [112] for the file. Each file has a unique GUID, which is assigned once and not re-used with very high probability. The LFN is a human-friendly name of the file, which may be different for each dataset containing the file. An usage example and additional details are presented below.

- `delete_files(dsn, list_of_files)`: Deletes logical files from the dataset with name `dsn`. The dataset state must be `OPEN`. The `list_of_files` is a list with either GUIDs or LFNs, because both can uniquely identify a file within a dataset as discussed below.

- `close_dataset(dsn)`: Sets the dataset with name `dsn` from state `OPEN` to state `CLOSED`. At this point, no files can be added or removed from the dataset, except if it is re-opened or a new version created.

- `open_dataset(dsn)`: Sets the dataset with name `dsn` from state `CLOSED` to state `OPEN` so that the contents of the latest dataset version can be modified.

- `freeze_dataset(dsn)`: Sets the dataset with name `dsn` from state `OPEN` or `CLOSED` to state `FROZEN`.

[1] Also referenced as "Universally Unique IDentifier" (UUID) or "Globally Unique IDentifier" (GUID).

- `create_version(dsn)`: Creates a new version for the dataset with name `dsn`. The dataset state must be `OPEN` or `CLOSED`. The version number, which is an integer set at dataset creation time to `version 1`, is incremented by `1`. The new dataset version includes the same files as the latest version; the dataset state is reset to `OPEN`.

- `list_files(dsn[, version]`: Lists the files in the latest version of the dataset with name `dsn`. If the optional attribute `version` is set, the method lists the files on the specific dataset version instead of the latest version.

- `list_datasets(pattern)`: Returns all dataset names that match the given pattern.

- `get_dataset_attributes(dsn)`: Returns the system attributes for the dataset with name `dsn`. System attributes include creation date, last modification date, dataset owner, number of files in dataset and total dataset size.

- `delete_dataset(dsn)`: Deletes the entire dataset definition for the dataset with name `dsn` from the catalogue.

Appendix B

Brief Description of the SRM v2.2 API

The storage services use the following subset of the SRM v2.2 methods:

- `srmPing`: Used to check if the service is operational.
- `srmPrepareToGet`: Used for preparing a file for transfer or access. The method assigns TURLs for each requested SURL, and allows the requester to specify a desired lifetime for the TURLs. It does not immediately return prepared files, since these operations can take some time. Instead, the method `srmStatusOfGetRequest` is used to check whether the files are prepared for access.
- `srmStatusOfGetRequest`: This function checks the status of the previously requested `srmPrepareToGet`.
- `srmReleaseFiles`: This releases the provided TURLs, after the read operation is concluded by the user.
- `srmLs`: Used to check whether files and directories exist, and return their metadata attributes. These attributes include the locality of the files, which may be one of `ONLINE`, `NEARLINE`, `ONLINE_AND_NEARLINE`, `LOST`, `NONE` or `UNAVAILABLE`.
- `srmMkdir`: Used to create directories in the storage namespace.
- `srmPrepareToPut`: Used to write files into the storage. Upon the client request, the SRM prepares a TURL so that the client can write data into that TURL. A lifetime is assigned on the TURL. The TURLs are not immediately provided because space may need to be allocated (e.g. copies that were left in a disk from previous requests may need to be removed first to make free space). The method `srmStatusOfPutRequest` is used to check whether the TURLs are prepared.

- `srmStatusOfPutRequest`: Used to check whether the files are prepared for write.
- `srmPutDone`: This method follows from the `srmStatusOfPutRequest` sequence, to indicate that the file has been fully written by the user into the provided TURL.
- `srmCopy`: Where supported, the method allows for third-party transfers between storages. More details are described in Section 5.3.4.1.
- `srmBringOnline`: This method brings files online upon the client request so that client can make certain data readily available for future access. In hierarchical storage systems, this method brings the files to the top hierarchy and requests the storage system to make sure that the files stay `ONLINE` for a specified period of time. It is equivalent to the `srmPrepareToGet` method, but does not immediately provide TURLs: it is used to anticipate access to files.
- `srmRm`: This function removes SURLs (the name space entries) from the storage.

Appendix C

Simulator

This appendix includes relevant code excerpts from the simulator described in Chapter 6. The code uses the following external libraries:

- **SciPy/NumPy**, a package for scientific computing with Python, available from *http://www.scipy.org/*.
- **RPy**, a Python interface to the R Programming Language, available from *http://rpy.sourceforge.net/*.
- **PyEM**, a Python module that implements the EM and BIC algorithms, developed by David Cournapeau and available from *http://www.ar.media.kyoto-u.ac.jp/members/david/softwares/em/index.html*.
- **SimPy**, an object-oriented process-based discrete-event simulation language for Python, available from *http://simpy.sourceforge.net/*.

The imported external Python modules are:

```
# Modules for NumPy/SciPy and R
import numpy
import pylab
import rpy
import scipy
import scipy.stats

# David Cournapeau's PyEM module
from em import GM, GMM, EM

# SimPy Discrete Event Simulator
from SimPy.Simulation import *
```

The following excerpt is the class that applies EM algorithm with Bayesian Information Clustering. The input data is passed in `matrix`.

```
class Learn_GM_BIC(object):
    def __init__(self, dimension, kmax, matrix, max_iterations=30, threshold=1e-10):
        lgm = []
        bics = numpy.zeros(kmax)
        em = EM()
        for i in xrange(0, kmax):
            lgm.append(GM(dimension, i+1, 'diag'))
            gmm = GMM(lgm[i], 'kmean')
            em.train(matrix, gmm, maxiter=max_iterations, thresh=threshold)
            bics[i] = gmm.bic(matrix)

        self.__best_lgm = lgm[numpy.argmax(bics)]
        self.__iterator = self.__samples()

    def __samples(self):
        while True:
            val = self.__best_lgm.sample(1)[0][0]
            if val > 0:
                yield val

    def sample(self):
        # Generate random sample
        return self.__iterator.next()

    def gaussian(self):
        return self.__best_lgm
```

The following class contains the code that predicts the duration of successful file transfers, given the input file size. This includes the instantiation of the previous class `Learn_GM_BIC` and the call that reads the historical workload from the relational database records in method `__get_durations`.

```
class DurationPredictorByFilesize(object):
    def __init__(self, run, channel):
        self.__run = run
        self.__channel = channel

        self.__predictors = {}
        filesize_durations = self.__get_durations()
        for typical_filesize in filesize_durations:
            durations = filesize_durations[typical_filesize]
            if len(durations) >= MIN_SUCCESS_SAMPLES: # skip if not enough samples
```

```
                matrix = numpy.array(map(lambda d: numpy.array([d])
                learn = Learn_GM_BIC(1,
                                     MAX_DURATION_PREDICTOR_CLUSTERS,
                                     matrix,
                                     durations)))
                self.__predictors[typical_filesize] = learn

    def __get_durations(self):
        # Read Workload from historical data.
        durations = {}
        for row in select("""SELECT duration, filesize, time
                                FROM """+TABLE_NAME+"""
                                WHERE runid=%s
                                AND channel=%s
                                AND service=%s
                                AND phase=%s
                                AND action=%s
                                AND success=%s
                                AND filesize>=%s""",
                            (self.__run,
                            self.__channel,
                            SERVICE_TRANS,
                            PHASE_TRANS,
                            ACTION_END,
                            True,
                            MIN_FILESIZE)):
            duration, filesize, evtime = row
            evtime = _to_epoch(evtime)

            typical_filesize = get_typical_size(filesize)
            durations.setdefault(typical_filesize, [])
            durations[typical_filesize].append(long(duration))
        return durations

    def sample(self, filesize):
        typical_filesize = get_typical_size(filesize)
        if typical_filesize in self.__predictors:
            return self.__predictors[typical_filesize].sample()
        # use closest size available!
        closest_size = get_closest_size(filesize, self.__predictors.keys())
        return self.__predictors[closest_size].sample()
```

The following SimPy process class is responsible for simulating each transfer. It reads transfer requests from a job queue (`jobq`) and uses the `DurationPredictorByFilesize` class for predicting the duration (in seconds) of successful transfers.

```
class TransferWorker(Process):
    def __init__(self,
                 jobq,
                 predictor_success,
                 bin_size, st_date,
                 bins, success_duration_filesize):
        Process.__init__(self)
        # input
        self.__jobq = jobq
        # parameters
        self.__predictor_success = predictor_success
        # output
        self.__bin_size = bin_size
        self.__st_date = st_date
        self.__bins = bins
        self.__success_duration_filesize = success_duration_filesize

    def execute(self):
        while True:
            job = self.__jobq.get()
            if not job:
                yield hold, self, 1
                continue

            # process job
            filesize = job
            typical_filesize = get_typical_size(filesize)

            # transfer success
            duration = self.__predictor_success.sample(filesize)

            # sleep until the transfer is finished
            yield hold, self, duration

            cur = (int(now())-self.__st_date)/self.__bin_size

            self.__bins[cur][0] += filesize
            self.__bins[cur][1] += 1
            self.__success_duration_filesize.setdefault(typical_filesize, [])
            self.__success_duration_filesize[typical_filesize].append(duration)
```

Bibliography

[1] A. Aamnitchi, S. Doraimani, and G. Garzoglio. Filecules in High-Energy Physics: Characteristics and Impact on Resource Management. In *Proceedings of the 15th International Symposium on High Performance Distributed Computing*, pages 69–80. IEEE Computer Society, 2006.

[2] David Adams, Dario Barberis, Chris Bee, Richard Hawkings, Sverre Jarp, Roger Jones, David Malon, Luc Poggioli, Gilbert Poulard, David Quarrie, and Torre Wenaus. The Atlas Computing Model. http://cdsweb.cern.ch/record/811058, 2004.

[3] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. *ACM SIGOPS Operating Systems Review*, 36(SI):1–14, 2002.

[4] R. Alfieri, R. Cecchini, V. Ciaschini, L. dell'Agnello, Á. Frohner, A. Gianoli, K. L orentey, and F. Spataro. Voms, an Authorization System for Virtual Organizations. In *Proceedings of the 1st European Across Grids Conference*, volume 2970/2004 of *Lecture Notes in Computer Science*, pages 33–40. Springer, 2004.

[5] Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Data Management and Transfer in High-Performance Computational Grid Environments. *Parallel Computing*, 28(5):749–771, 2002.

[6] William Allcock, John Bresnahan, Rajkumar Kettimuthu, and Michael Link. The Globus Striped GridFTP Framework and Server. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing.* IEEE Computer Society, 2005.

[7] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 562–570. IEEE Computer Society, 1976.

[8] Chris Anderson. The End of Theory: The Data Deluge Makes the Scientific Method Obsolete, 2008. Wired Magazine, http://www.wired.com/science/discoveries/magazine/16-07/pb_theory.

[9] Phil Andrews, Patricia Kovatch, and Christopher Jordan. Massive High-Performance Global File Systems for Grid computing. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing.* IEEE Computer Society, 2005.

[10] Mario Antonioletti, Malcolm Atkinson, Rob Baxter, Andrew Borley, Neil P. Chue Hong, Brian Collins, Neil Hardman, Alastair C. Hume, Alan Knox, Mike Jackson, Amy Krause, Simon Laws, James Magowan, Norman W. Paton, Dave Pearson, Tom Sugden, Paul Watson, and Martin Westhead. The Design and Implementation of Grid Database Services in OGSA-DAI: Research Articles. *Concurrency and Computation: Practice & Experience*, 17(2-4):357–376, 2005.

[11] James Aspnes, Yossi Azar, Amos Fiat, Serge Plotkin, and Orli Waarts. On-Line Routing of Virtual Circuits with Applications to Load Balancing and Machine Scheduling. *Journal of the ACM*, 44(3):486–504, 1997.

[12] Baruch Awerbuch and Yossi Azar. Local Optimization of Global Objectives: Competitive Distributed Deadlock Resolution and Resource Allocation. In *Proceedings of the 35th IEEE Symposium on Foundations of Computer Science*, pages 240–249. IEEE Computer Society, 1994.

[13] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced Allocations (extended abstract). In *Proceedings of the 26th ACM Symposium on Theory of computing*, pages 593–602. ACM, 1994.

[14] Yossi Azar, Edith Cohen, Amos Fiat, Haim Kaplan, and Harald Räcke. Optimal Oblivious Routing in Polynomial Time. *Journal of Computer and System Sciences*, 69(3):383–394, 2004.

[15] Ricardo Baeza-Yates and Raghu Ramakrishnan. Data challenges at Yahoo! In *Proceedings of the 11th International Conference on Extending Database Technology*, pages 652–655. ACM, 2008.

[16] Olof Bärring, Ben Couturier, Jean-Damien Durand, Emil Knezo, Sebastien Ponce, and Vitaly Motyakov. Storage Resource Sharing with CASTOR. In *Proceedings of the 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, 2004.

[17] Chaitanya Baru, Reagan Moore, Arcot Rajasekar, and Michael Wan. The SDSC Storage Resource Broker. In *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative Research*. IBM Press, 1998.

[18] Sujoy Basu, Sujata Banerjee, Puneet Sharma, and Sung-Ju Lee. NodeWiz: Peer-to-peer Resource Discovery for Grids. In *Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid*, volume 1, pages 213–220. IEEE Computer Society, 2005.

[19] J-P. Baud, J. Casey, S. Lemaitre, C. Nicholson, D. Smith, and G. Stewart. Lcg Data Management: From EDG to EGEE. In *UK eScience All Hands Meeting Proceedings*, Nottingham, UK, 2005.

[20] William H. Bell, David G. Cameron, Luigi Capozza, A. Paul Millar, Kurt Stockinger, and Floriano Zini. Simulation of Dynamic Grid Replication Strategies in OptorSim. In *Proceedings of the 3rd International Workshop on Grid Computing*, volume 2536 of *Lecture Notes in Computer Science*, pages 46–57. Springer, 2002.

[21] William H. Bell, David G. Cameron, Ruben Carvajal-Schiaffino, A. Paul Millar, Kurt Stockinger, and Floriano Zini. Evaluation of an Economy-Based File Replication Strategy for a Data Grid. In *Proceedings of the 3rd IEEE International Symposium on Cluster Computing and the Grid*, pages 661–668. IEEE Computer Society, 2003.

[22] Tim Berners-Lee, Roy Thomas Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. RFC 2396 (Draft Standard), 1998.

[23] Joseph Bester, Ian Foster, Carl Kesselman, Jean Tedesco, and Steven Tuecke. GASS: A Data Movement and Access Service for Wide Area Computing Systems. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 78–88. ACM, 1999.

[24] Ranjita Bhagwan, Kiran Tati, Yu Cheng, Stefan Savage, and Geoff Voelker. Total Recall: System Support for Automated Availability Management. In *Proceedings of 1st USENIX Symposium on Networked Systems Design and Implementation.* USENIX Association, 2004.

[25] Charles Blake and Rodrigo Rodrigues. High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two. In *Proceedings of the 9th conference on Hot Topics in Operating Systems*, volume 9. USENIX Association, 2003.

[26] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP) 1.1 - NOTE-SOAP-20000508. World Wide Web Consortium, 2000.

[27] Peter Braam. File Systems for Clusters from a Protocol Perspective. In *Proceedings of Second Extreme Linux Topics Workshop.* USENIX Association, 1999.

[28] Miguel Branco and Luc Moreau. Enabling Provenance on Large Scale e-Science Applications. In *Provenance and Annotation of Data*, volume 4145 of *Lecture Notes in Computer Science*, pages 55–63. Springer, 2006.

[29] Miguel Branco, Ed Zaluska, David De Roure, Mario Lassnig, and Vincent Garonne. Managing very large distributed datasets on a Data Grid. *Concurrency and Computation: Practice & Experience (in press)*, 2009.

[30] Miguel Branco, Ed Zaluska, David De Roure, Pedro Salgado, Vincent Garonne, Mario Lassnig, and Ricardo Rocha. Managing Very-Large Distributed Datasets. In *Proceedings of the OTM Conferences*, volume 5331/2008 of *Lecture Notes in Computer Science*, pages 775–792. Springer, 2008.

[31] Lee Breslau, Pei Cue, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 1, pages 126–134. IEEE Computer Society, 1999.

[32] John Bresnahan, Michael Link, Gaurav Khanna, Zulfikar Imani, Rajkumar Kettimuthu, and Ian Foster. Globus GridFTP: What's New in 2007. In *Proceedings of the 1st International Conference on Networks for Grid Applications*, pages 1–5. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007.

[33] Mike Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 335–350. USENIX Association, 2006.

[34] Ali Butt, Troy Johnson, Yili Zheng, and Y. Hu. Kosha: a Peer-to-Peer Enhancement for the Network File System. *Journal of Grid Computing*, 4(3):323–341, 2006.

[35] Min Cai, Ann Chervenak, and Martin Frank. A Peer-to-Peer Replica Location Service Based on A Distributed Hash Table. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2004.

[36] D. G. Cameron, A. P. Millar, C. Nicholson, R. Carvajal-Schiaffino, K. Stockinger, and F. Zini. Analysis of Scheduling and Replica Optimisation Strategies for Data Grids Using OptorSim. *Journal of Grid Computing*, 2(1):57–69, 2004.

[37] David G. Cameron, Ruben Carvajal-Schiaffino, A. Paul Millar, Caitriana Nicholson, Kurt Stockinger, and Floriano Zini. Evaluating Scheduling and Replica Optimisation Strategies in OptorSim. In *Proceedings of the 4th International Workshop on Grid Computing*, pages 52–59. IEEE Computer Society, 2003.

[38] Henri Casanova, Arnaud Legrand, and Martin Quinson. SimGrid: A Generic Framework for Large-Scale Distributed Experiments. In *Proceedings of the 10th International Conference on Computer Modeling and Simulation*, pages 126–131. IEEE Computer Society, 2008.

[39] Miguel Castro, Manuel Costa, and Antony Rowstron. Debunking some myths about structured and unstructured overlays. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation*, volume 2, pages 85–98. USENIX Association, 2005.

[40] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 173–186. USENIX Association, 1999.

[41] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2):1–26, 2008.

[42] Ann Chervenak, Ewa Deelman, Ian Foster, Leanne Guy, Wolfgang Hoschek, Adriana Iamnitchi, Carl Kesselman, Peter Kunszt, Matei Ripeanu, Bob Schwartzkopf, Heinz Stockinger, Kurt Stockinger, and Brian Tierney. Giggle: A Framework for Constructing Scalable Replica Location Services. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–17. IEEE Computer Society, 2002.

[43] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets. *Journal of Network and Computer Applications*, 23(3):187–200, 2000.

[44] Ann Chervenak, Robert Schuler, Carl Kesselman, Scott Koranda, and Brian Moe. Wide Area Data Replication for Scientific Collaborations. *International Journal of High Performance Computing and Networking*, 5(3):124–134, 2008.

[45] Hung chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 1029–1040. ACM, 2007.

[46] Byung-Gon Chun, Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, M. Frans Kaashoek, John Kubiatowicz, and Robert Morris. Efficient Replica Maintenance for Distributed Storage Systems. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, volume 3, pages 45–58. USENIX Association, 2006.

[47] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Proceedings of the ICSI International Workshop on Design Issues in Anonymity and Unobservability*, volume 2009/2001 of *Lecture Notes in Computer Science*, pages 46–66. Springer, 2001.

[48] Bram Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, 2003.

[49] Edith Cohen and Scott Shenker. Replication Strategies in Unstructured Peer-to-Peer Networks. In *Proceedings of the 2002 ACM SIGCOMM Conference*, pages 177–190. ACM, 2002.

[50] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[51] Karl Czajkowski, Carl Kesselman, Steven Fitzgerald, and Ian Foster. Grid Information Services for Distributed Resource Sharing. In *Proceedings of the 10th IEEE International Symposium on High-Performance Distributed Computing*. IEEE Computer Society, 2001.

[52] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. *ACM SIGOPS Operating Systems Review*, 35(5):202–215, 2001.

[53] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, Frans Kaashoek, and Robert Morris. Designing a DHT for Low Latency and High Throughput. In *Proceedings of 1st USENIX Symposium on Networked Systems Design and Implementation*, pages 85–98. USENIX Association, 2004.

[54] Miguel de Cervantes. *El ingenioso hidalgo don Quijote de la Mancha*. Originally published by Iuan de la Cuesta, 1605.

[55] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2004.

[56] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[57] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 205–220. ACM, 2007.

[58] Ewa Deelman and Ann Chervenak. Data Management Challenges of Data-Intensive Scientific Workflows. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid*, pages 687–692. IEEE Computer Society, 2008.

[59] Ewa Deelman, Gurmeet Singh, Miron Livny, Bruce Berriman, and John Good. The Cost of Doing Science on the Cloud: The Montage Example. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12. IEEE Computer Society, 2008.

[60] Kemal A. Delic and Martin Anthony Walker. Emergence of The Academic Computing Clouds. *Ubiquity*, 9(31):1–1, 2008.

[61] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society*, 39(1):1–38, 1977.

[62] Mustafa Mat Deris, Jemal H. Abawajy, and Ali Mamat. An efficient replicated data access approach for large-scale distributed systems. *Future Generation Computer Systems*, 24(1):1–9, 2008.

[63] P. Deutsch and J-L. Gailly. ZLIB Compressed Data Format Specification version 3.3. RFC 1950 (Informational), 1996.

[64] Shyamala Doraimani and Adriana Iamnitchi. File grouping for scientific data management: lessons from experimenting with real traces. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, pages 153–164. ACM, 2008.

[65] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, pages 75–80. USENIX Association, 2001.

[66] L. Dusseault. HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV). RFC 4918 (Proposed Standard), 2007.

[67] The Economist. Microsoft v Google: When clouds collide. http://www.economist.com/business/displaystory.cfm?story_id=10650607, February 7th 2008.

[68] Giacomo V. Mc Evoy and Bruno Schulze. Using Clouds to address Grid Limitations. In *Proceedings of the 6th International Workshop on Middleware for Grid computing*, pages 1–6. ACM, 2008.

[69] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[70] Roy Thomas Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), 1999.

[71] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[72] Ian Foster, K. Czajkowski, D. E. Ferguson, J. Frey, S. Graham, T. Maguire, D. Snelling, and S. Tuecke. Modeling and Managing State in Distributed Systems: The Role of OGSI and WSRF. *Proceedings of the IEEE*, 93(3):604–612, 2005.

[73] Ian Foster and Adriana Iamnitchi. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In *Peer-to-Peer Systems II*, volume 2735 of *Lecture Notes in Computer Science*, pages 118–128. Springer, 2003.

[74] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2nd edition, 2003.

[75] Ian Foster, Carl Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Open Grid Service Infrastructure WG, 2002.

[76] Ian Foster, Carl Kesselman, Gene Tsudik, and Steven Tuecke. A Security Architecture for Computational Grids. In *Proceedings of the 5th ACM conference on Computer and Communications Security*, pages 83–92. ACM, 1998.

[77] Ian T. Foster. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. In *Proceedings of the 7th International Conference on Parallel Processing*, pages 1–4. Springer, 2001.

[78] David Freedman and Persi Diaconis. On the histogram as a density estimator: l_2 theory. *Probability Theory and Related Fields*, 57:453–476, 1981.

[79] Patrick Fuhrmann and Volker Gülzow. dcache, Storage System for the Future. In *Euro-Par 2006 Parallel Processing*, volume 4128 of *Lecture Notes in Computer Science*, pages 1106–1113. Springer, 2006.

[80] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *ACM SIGOPS Operating Systems Review*, 37(5):29–43, 2003.

[81] Geof H. Givens and Jennifer A. Hoeting. *Computational Statistics*. Wiley-Interscience, 1st edition, 2005.

[82] P. Brighten Godfrey, Scott Shenker, and Ion Stoica. Minimizing Churn in Distributed Systems. In *Proceedings of the 2006 ACM SIGCOMM Conference*, pages 147–158. ACM, 2006.

[83] Jim Gray and Leslie Lamport. Consensus on Transaction Commit. *ACM Transactions on Database Systems*, 31(1):133–160, 2006.

[84] Jim Gray, David T. Liu, Maria Nieto-Santisteban, Alex Szalay, David J. DeWitt, and Gerd Heber. Scientific Data Management in the Coming Decade. *ACM SIGMOD Record*, 34(4):34–41, 2005.

[85] Alon Halevy, Peter Norvig, and Fernando Pereira. The Unreasonable Effectiveness of Data. *IEEE Intelligent Systems*, 24(2):8–12, 2009.

[86] Brian Hayes. Cloud computing. *Communications of the ACM*, 51(7):9–11, 2008.

[87] Joseph M. Hellerstein. Toward Network Data Independence. *ACM SIGMOD Record*, 32(3):34–40, 2003.

[88] Tony Hey and Anne Trefethen. The data deluge: An e-science perspective. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, chapter 36. Wiley, 2003.

[89] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.

[90] Wilson Hsieh, Jayant Madhavan, and Rob Pike. Data management projects at Google. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 725–726. ACM, 2006.

[91] IPOQUE. Internet Study 2007: Data about P2P, VoIP, Skype, file hosters like RapidShare and streaming services like YouTube. http://www.ipoque.com/media/internet_studies/internet_study_2007, 2007.

[92] R. K. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley-Interscience, 1991.

[93] Dieter Jungnickel. *Graphs, Networks and Algorithms.* Springer, 2nd edition, 2003.

[94] G. Khanna, U. Catalyurek, T. Kurc, R. Kettimuthu, P. Sadayappan, and J. Saltz. A Dynamic Scheduling Approach for Coordinated Wide-Area Data Transfers using GridFTP. In *Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12. IEEE Computer Society, 2008.

[95] Gaurav Khanna, Umit Catalyurek, Tahsin Kurc, Rajkumar Kettimuthu, P. Sadayappan, Ian Foster, and Joel Saltz. Using Overlays for Efficient Data Transfer Over Shared Wide-Area Networks. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12. IEEE Computer Society, 2008.

[96] Gaurav Khanna, Umit Catalyurek, Tahsin Kurc, P. Sadayappan, Joel Saltz, Rajkumar Kettimuthu, and Ian Foster. Multi-Hop Path Splitting and Multi-Pathing Optimizations for Data Transfers over Shared Wide-Area Networks using GridFTP. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, pages 225–226. ACM, 2008.

[97] J. Klensin. Role of the Domain Name System (DNS). RFC 3467 (Informational), 2003.

[98] John C. Klensin. Simple Mail Transfer Protocol. RFC 5321 (Draft Standard), 2008.

[99] Donald Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, 3rd edition, 1997.

[100] Tevfik Kosar and Miron Livny. Stork: Making Data Placement a First Class Citizen in the Grid. In *Proceedings of the 24th International Conference on Distributed Computing Systems*, pages 342–349. IEEE Computer Society, 2004.

[101] Nicolas Kourtellis, Lydia Prieto, Adriana Iamnitchi, Gustavo Zarrate, and Dan Fraser. Data Transfers in the Grid: Workload Analysis of Globus GridFTP. In *Proceedings of the 2008 International Workshop on Data-aware Distributed Computing*, pages 29–38. ACM, 2008.

[102] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. *ACM SIGPLAN Notices*, 35(11):190–201, 2000.

[103] P. Kunszt, P. Badino, A. Frohner, G. McCance, K. Nienartowicz, R. Rocha, and D. Rodrigues. Data Storage, Access and Catalogs in gLite. In *Proceedings of the 2005 IEEE International Symposium on Mass Storage Systems and Technology*, volume 0, pages 166–170. IEEE Computer Society, 2005.

[104] Peter Kunszt, Erwin Laure, Heinz Stockinger, and Kurt Stockinger. File-based replica management. *Future Generation Computer Systems*, 21(1):115–123, 2005.

[105] H. Lamehamedi, B. Szymanski, Z. Shentu, and E. Deelman. Data Replication Strategies in Grid Environments. In *Proceedings of the 5th International Conference on Algorithms and Architectures for Parallel Processing*, pages 378–383. IEEE Computer Society, 2002.

[106] Houda Lamehamedi, Zujun Shentu, Boleslaw Szymanski, and Ewa Deelman. Simulation of Dynamic Data Replication Strategies in Data Grids. In *Proceedings of the 17th IEEE International Symposium on Parallel and Distributed Processing.* IEEE Computer Society, 2003.

[107] Houda Lamehamedi, Boleslaw K. Szymanski, and Brenden Conte. Distributed Data Management Services for Dynamic Data Grids. Technical report, Rensselaer Polytechnic Institute, 2005.

[108] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[109] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):18–25, 2001.

[110] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[111] Simon St. Laurent, Edd Dumbill, and Joe Johnston. *Programming Web Services with XML-RPC.* O'Reilly Media, 2001.

[112] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122 (Proposed Standard), 2005.

[113] Ming Lei, Susan V. Vrbsky, and Xiaoyan Hong. An on-line replication strategy to increase availability in Data Grids. *Future Generation Computer Systems*, 24(2):85–98, 2008.

[114] Ming Lei, S.V. Vrbsky, and Xiaoyan Hong. A Dynamic Data Grid Replication Strategy to Minimize the Data Missed. In *Proceedings of the 3rd International Conference on Broadband Communications, Networks and Systems*, pages 1–10. IEEE Computer Society, 2006.

[115] Elias Leontiadis, Vassilios Dimakopoulos, and Evaggelia Pitoura. Creating and Maintaining Replicas in Unstructured Peer-to-Peer Systems. In *Euro-Par 2006 Parallel Processing*, volume 4128/2006 of *Lecture Notes in Computer Science*, pages 1015–1025. Springer, 2006.

[116] Yi-Fang Lin, Pangfeng Liu, and Jan-Jan Wu. Optimal Placement of Replicas in Data Grid Environments with Locality Assurance. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems*, volume 1. IEEE Computer Society, 2006.

[117] Witold Litwin. *Linear Hashing: a new tool for file and table addressing.*, pages 570–581. Morgan Kaufmann Publishers Inc., 1988.

[118] Pangfeng Liu and Jan-Jan Wu. Optimal Replica Placement Strategy for Hierarchical Data Grid Systems. In *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid*, volume 1. IEEE Computer Society, 2006.

[119] Dionysios Logothetis and Kenneth Yocum. Ad-Hoc Data Processing in the Cloud. *Journal of the VLDB Endowment*, 1(2):1472–1475, 2008.

[120] Eng Keong Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A Survey and Comparison of Peer-to-Peer Overlay Network Schemes. *IEEE Communications Surveys & Tutorials*, 7(2):72–93, 2005.

[121] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and Replication in Unstructured Peer-to-Peer Networks. In *Proceedings of the 16th International Conference on Supercomputing*, pages 84–95. ACM, 2002.

[122] John MacCormick, Nicholas Murphy, Venugopalan Ramasubramanian, Udi Wieder, Junfeng Yang, and Lidong Zhou. Kinesis: A New Approach to Replica Placement in Distributed Storage Systems. *ACM Transactions on Storage*, 4(4):1–28, 2009.

[123] Y. Machida, S. Takizawa, H. Nakada, and S. Matsuoka. Multi-Replication with Intelligent Staging in Data-Intensive Grid Applications. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, pages 88–95. IEEE Computer Society, 2006.

[124] Ravi K. Madduri, Cynthia S. Hood, and William E. Allcock. Reliable File Transfer in Grid Environments. In *Proceedings of the 27th Annual IEEE Conference on Local Computer Networks*, pages 737–738. IEEE Computer Society, 2002.

[125] Tadashi Maeno. PanDA: distributed production and distributed analysis system for ATLAS. *Journal of Physics: Conference Series*, 119(6), 2008.

[126] Frank Manola and Eric Miller. RDF Primer. World Wide Web Consortium, 2004.

[127] Ralph Merkle. *Secrecy, authentication and public key systems / A certified digital signature.* PhD thesis, Stanford University, 1979.

[128] Sun Microsystems. Lustre Networking: High-Performance Features and Flexible Support for a Wide Array of Networks. http://www.sun.com/offers/details/lustre_networking.html, 2008.

[129] Ruggero Morselli, Bobby Bhattacharjee, Aravind Srinivasan, and Michael A. Marsh. Efficient Lookup on Unstructured Topologies. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, pages 77–86. ACM, 2005.

[130] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: a Read-/Write Peer-to-Peer File System. *ACM SIGOPS Operating Systems Review*, 36(SI):31–44, 2002.

[131] Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay V. Vazirani. *Algorithmic Game Theory*. Cambridge University Press, 2007.

[132] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1099–1110. ACM, 2008.

[133] Ekow Otoo and Arie Shoshani. Accurate Modeling of Cache Replacement Policies in a Data Grid. In *Proceedings of the 20th NASA Goddard Conference on Mass Storage Systems and Technologies*, page 10. IEEE Computer Society, 2003.

[134] Mayur R. Palankar, Adriana Iamnitchi, Matei Ripeanu, and Simson Garfinkel. Amazon S3 for Science Grids: a Viable Solution? In *Proceedings of the 2008 International Workshop on Data-aware Distributed Computing*, pages 55–64. ACM, 2008.

[135] LIGO Project. Lightweight Data Replicator. http://www.lsc-group.phys.uwm.edu/LDR/, 2004.

[136] Harald Räcke. Minimizing Congestion in General Networks. In *Proceedings of the 43rd IEEE Symposium on Foundations of Computer Science*, pages 43–52. IEEE Computer Society, 2002.

[137] Harald Räcke. Optimal Hierarchical Decompositions for Congestion Minimization in Networks. In *Proceedings of the 40th ACM Symposium on Theory of computing*, pages 255–264. ACM, 2008.

[138] Ioan Raicu, Yong Zhao, Ian T. Foster, and Alex Szalay. Accelerating Large-scale Data Exploration through Data Diffusion. In *Proceedings of the 2008 International Workshop on Data-aware Distributed Computing*, pages 9–18. ACM, 2008.

[139] A. Rajasekar, M. Wan, R. Moore, and W. Schroeder. Data Grid federation. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 541–546. CSREA Press, 2004.

[140] A. Rajasekar, M. Wan, R. Moore, and W. Schroeder. A Prototype Rule-Based Distributed Data Management System. In *HPDC workshop on Next Generation Distributed Data Management*, 2006.

[141] Kavitha Ranganathan and Ian Foster. Design and Evaluation of Dynamic Replication Strategies for a High Performance Data Grid. In *International Conference on Computing in High Energy and Nuclear Physics*. IOP, 2001.

[142] Kavitha Ranganathan and Ian Foster. Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications. In *Proceedings of the 11th International Symposium on High Performance Distributed Computing*, pages 352–358. IEEE Computer Society, 2002.

[143] Kavitha Ranganathan and Ian Foster. Simulation Studies of Computation and Data Scheduling Algorithms for Data Grids. *Journal of Grid Computing*, 1(1):53–62, 2003.

[144] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A Scalable Content-Addressable Network. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 161–172. ACM, 2001.

[145] J. Rehn, T. Barrass, D. Bonacorsi, J. Hernandez, I. Semeniouk, L. Tuura, and Y. Wu. PhEDEx high-throughput data transfer management system. In *International Conference on Computing in High Energy and Nuclear Physics*, 2006.

[146] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: The OceanStore Prototype. In *Proceedings of the 2003 Conference on File and Storage Technologies*, pages 1–14. USENIX Association, 2003.

[147] John Risson and Tim Moors. Survey of Research towards Robust Peer-to-Peer networks: Search Methods. *Computer Networks*, 50(17):3485–3521, 2006.

[148] Mema Roussopoulos, Mary Baker, David S. H. Rosenthal, T.J. Giuli, Petros Maniatis, and Jeff Mogul. 2 P2P or Not 2 P2P? *CoRR*, cs.NI/0311017, 2003.

[149] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, volume 2218/2001 of *Lecture Notes in Computer Science*, pages 329–350. Springer, 2001.

[150] Asad Samar and Heinz Stockinger. Grid Data Management Pilot (GDMP): A Tool for Wide Area Replication. In *Proceedings of the IASTED International Conference on Applied Informatics*, 2001.

[151] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, pages 119–130. USENIX Association, 1985.

[152] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[153] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 2002 Conference on File and Storage Technologies*, pages 231–244. USENIX Association, 2002.

[154] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[155] Philip Schwan. Lustre: Building a File System for 1,000-node Clusters. In *Proceedings of the 2003 Linux Symposium*, 2003.

[156] Gideon Schwarz. Estimating the Dimension of a Model. *The Annals of Statistics*, 6(2):461–464, 1978.

[157] A. Shoshani, A. Sim, and J. Gu. Storage Resource Managers: Middleware Components for Grid Storage. In *Proceedings of the 10th NASA Goddard Conference on Mass Storage Systems and Technologies*. IEEE Computer Society, 2002.

[158] Stephen C. Simms, Gregory G. Pike, S. Teige, Bret Hammond, Yu Ma, Larry L. Simms, C. Westneat, and Douglas A. Balog. Empowering Distributed Workflow with the Data Capacitor: Maximizing Lustre Performance across the Wide Area Network. In *Proceedings of the 2007 Workshop on Service-Oriented Computing Performance: Aspects, Issues, and Approaches*, pages 53–58. ACM, 2007.

[159] Mauro Sozio, Thomas Neumann, and Gerhard Weikum. Near-Optimal Dynamic Replication in Unstructured Peer-to-Peer Networks. In *Proceedings of the 27th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 281–290. ACM, 2008.

[160] Heinz Stockinger, Asad Samar, Koen Holtman, Bill Allcock, Ian Foster, and Brian Tierney. File and Object Replication in Data Grids. *Cluster Computing*, 5(3):305–314, 2002.

[161] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160. ACM, 2001.

[162] Jeremy Stribling, Yair Sovran, Irene Zhang, Xavid Pretzer, Jinyang Li, M. Frans Kaashoek, and Robert Morris. Flexible, Wide-Area Storage for Distributed Systems with WheelFS. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, pages 43–58. USENIX Association, 2009.

[163] Anthony Sulistio, Uros Cibej, Srikumar Venugopal, Borut Robic, and Rajkumar Buyya. A toolkit for modelling and simulating data Grids: an extension to GridSim. *Concurrency and Computation: Practice & Experience*, 20(13):1591–1609, 2008.

[164] Alexander S. Szalay, Jim Gray, Ani R. Thakar, Peter Z. Kunszt, Tanu Malik, Jordan Raddick, Christopher Stoughton, and Jan vandenBerg. The SDSS SkyServer: Public Access to the Sloan Digital Sky Server data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 570–581. ACM, 2002.

[165] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Pearson Prentice Hall, 2nd edition, 2007.

[166] Osamu Tatebe, Satoshi Sekiguchi, and Youhei Morita. Gfarm v2: A grid file system that supports high-performance distributed and parallel data computing. http://datafarm.apgrid.org/pdf/CHEP04-gfarmv2.pdf, 2004.

[167] S. Tuecke, K. Czajkowski, Ian Foster, J. Frey, S. Graham, C. Kesselman, D. Snelling, and Vanderbilt. Open Grid Services Infrastructure (OGSI), 2003.

[168] S. Venugopal and R. Buyya. A Set Coverage-based Mapping Heuristic for Scheduling Distributed Data-Intensive Applications on Global Grids. In *Proceedings of 7th IEEE/ACM International Conference on Grid Computing*, pages 238–245. IEEE Computer Society, 2006.

[169] Srikumar Venugopal, Rajkumar Buyya, and Kotagiri Ramamohanarao. A Taxonomy of Data Grids for Distributed Data Sharing, Management, and Processing. *ACM Computing Surveys*, 38(1), 2006.

[170] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[171] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 307–320. USENIX Association, 2006.

[172] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 31. IEEE Computer Society, 2006.

[173] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters. In *Proceedings of the 2nd International Workshop on Petascale Data Storage*, pages 35–44. ACM, 2007.

[174] Aaron Weiss. Computing in the clouds. *netWorker*, 11(4):16–25, 2007.

[175] Brian S. White, Michael Walker, Marty Humphrey, and Andrew S. Grimshaw. LegionFS: A Secure and Scalable File System Supporting Cross-Domain High-Performance Applications. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 59–59. ACM, 2001.

[176] Nancy Wilkins-Diehr, Dennis Gannon, Gerhard Klimeck, Scott Oster, and Sudhakar Pamidighantam. TeraGrid Science Gateways and Their Impact on Science. *Computer*, 41(11):32–41, 2008.

[177] Chao-Tung Yang, I-Hsien Yang, Chun-Hsiang Chen, and Shih-Yu Wang. Implementation of a dynamic adjustment mechanism with efficient replica selection in data grid environments. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 797–804. ACM, 2006.

[178] Yulai Yuan, Yongwei Wu, Guangwen Yang, and Feng Yu. Dynamic Data Replication based on Local Optimization Principle in Data Grid. In *Proceedings of the 6th International Conference on Grid and Cooperative Computing*, pages 815–822. Springer, 2007.

[179] Zheng Zhang, Qiao Lian, Shiding Lin, Wei Chen, Yu Chen, and Chao Jin. BitVault: a Highly Reliable Distributed Data Retention Platform. *ACM SIGOPS Operating Systems Review*, 41(2):27–36, 2007.

[180] Ben Y. Zhao, John D.Kubiatowicz, and Anthony D. Joseph. Tapestry: A Fault-Tolerant Wide-area Application Infrastructure. *ACM SIGCOMM Computer Communication Review*, 32(1):81–81, 2002.