Managing very-large distributed datasets

Miguel Branco, Ed Zaluska, David de Roure, Pedro Salgado,
Vincent Garonne, Mario Lassnig, and Ricardo Rocha

CERN - European Organization for Nuclear Research,
University of Southampton, UK,
University of Innsbruck, Austria

Abstract. In this paper, we introduce a system for handling very large datasets, which need to be stored across multiple computing sites. Data distribution introduces complex management issues, particularly as computing sites may make use of different storage systems with different internal organizations. The motivation for our work is the ATLAS Experiment for the Large Hadron Collider (LHC) at CERN, where the authors are involved in developing the data management middleware. This middleware, called DQ2, is charged with shipping petabytes of data every month to research centers and universities worldwide and has achieved aggregate throughputs in excess of 1.5 Gbytes/sec over the wide-area network. We describe DQ2's design and implementation, which builds upon previous work on distributed file systems, peer-to-peer systems and Data Grids. We discuss its fault tolerance and scalability properties and briefly describe results from its daily usage for the ATLAS Experiment.

Key words: Data Management, Data Grids, Distributed Systems, Grid Computing, Datasets

1 Introduction

Our work addresses the problem of managing very large datasets. The motivation is the LHC project at CERN, which is expected to start operation during the summer of 2008 and continue in production for about twenty years. The LHC particle accelerator, extending for a 27 km ring buried 100 meters underground is illustrated in Figure 1, along with the various LHC detectors. The raw data produced by just one of the LHC detectors (the ATLAS Experiment [1]) exceeds ten petabytes per year. ATLAS is a worldwide collaboration that will produce petabytes of data during its lifetime. These data needs to be distributed and stored globally for access by a large number of scientists.

In this paper we start with a review of contributions in the area of distributed file systems, peer-to-peer and data grids. Based on this prior work, we propose a new architecture, improving the previous contributions in several respects. We describe important properties of the system and initial experiences running a real world production infrastructure for the ATLAS Experiment.

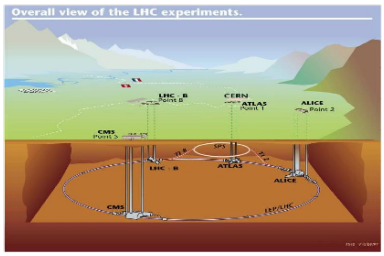


Fig. 1. Schematic overview of the LHC accelerator.

2 Existing Work

A number of different Computer Science areas have devised architectures and software systems to address the problem of very large datasets. Some of the most relevant areas are distributed file systems, peer-to-peer (P2P) systems and Data Grids. In this section we introduce the major contributions from these areas.

2.1 Distributed File Systems

NFS [10] is one of the early distributed file systems which 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. With large numbers of users or under 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 [9] was the first distributed file system to introduce client-side caching. This property increases the scalability of AFS but introduces additional complexity when handling updates. UNIX "last file write wins" semantics are hard to implement in a scalable manner. AFS introduced "last file close wins" semantics. This limits the universal applicability of AFS but increases its scalability for the common cases of multiple reads with infrequent writes. We are not aware of any AFS-based system handling petabytes of data over a wide-area network. We believe this is due to fundamental AFS design decisions, such assupporting POSIX-like semantics. Coda [15], a successor of AFS, introduced support for disconnected operations when the network connection is lost, but 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) [16]. GPFS provides high-performance I/O due to its ability to stripe blocks of data from individual files over multiple disks. GPFS has been demonstrated to work over a wide-area network [17] but under strict deployment constraints. Block-based systems can be expected to have difficulties scaling to very large user numbers, particularly in a shared environment.

The Lustre [21] distributed file system was originally developed by Cluster File Systems. Lustre is inspired by the architecture devised for the Digital VAX-Clusters, 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. Lustre scales to the data handling requirements discussed previously: tens of thousands of nodes and PetaBytes of storage. Lustre was designed as a cluster file system for a closed network but since has been expanding to accommodate multi-site and multi-cluster deployments. Lustre briefly described plans for a "Lustre Router Control Panel" to allow adjusting of quality of service within a cluster and wide-area network.

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.

Google has designed and implemented the Google File System (GFS) [11], 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 (each 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.

GFS can scale to large clusters while running on inexpensive commodity hardware. Hadoop[1], a top-level Apache project, is a system inspired by the design of GFS but open source and therefore considerably better documented. A core lesson from these systems is that scalability is achieved by taking advantage of environment constraints: for example, GFS eliminates the complex distributed locking models of earlier systems by allowing append operations only and adopting simple mechanisms for fault-tolerance.

2.2 Peer to Peer systems

Peer-to-peer (P2P) systems are particularly interesting for their scalability and ability to cope with heterogeneous environments. This has been an area of research with many contributions in the past years. Work described in [4] provides an excellent analysis of the search aspects for P2P systems.

There are several different architectures for P2P networks that may be classified in 'centralized', 'decentralized but structured' or 'decentralized and unstructured' [23]. While decentralized and unstructured systems were commonly used

[1] Refer to `http://hadoop.apache.org/core/`.

given their scalability but most have evolved to have associated structures, usually by relying on super-peer nodes or DHT algorithms [4]. This is an important lesson for our own architecture. P2P research has also analyzed the searching aspect in conjunction with storage and replication of data. Work by [23] and [14] has shown how to minimize exchange of messages between peers whilst providing effective mechanisms to locate data and decide on replication in peer nodes other than the data requestor, as to optimize future requests.

2.3 Data Grids

GASS [18] or "Global Access to Secondary Storage" is one of the early Grid Computing contributions to the large datasets problem. It consists of a system designed to manage secondary caches, which is 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 as in distributed file systems, but its functionality is very limited.

GDMP (Grid Data Mirroring Package) [20] is a file and object replication tool. It introduced the concept of a storage system subscribing to collections of files that were then moved using GridFTP [7]. GDMP was envisaged as a limited prototype system for file movement and its scalability was not investigated.

Ann Chernevak et al [2] introduced the "Data Grid" in an architecture paper which defines a specialized Grid architecture for handling large data volumes. The architecture is loosely defined to accommodate various models of operation but is tightly integrated with "Grid dynamics": security, awareness of virtual organizations 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. The paper also introduces higher-level services to integrate all the individual lower-level services onto a coherent set, defining a Replica Management service capable of moving files between Grid sites and doing all the necessary bookkeeping. In addition it also defined the Replica Selection and Filtering service that would decide on-demand replication.

OptorSim [3] is an example of a simulator built mainly to study how to optimize access to data from Grid jobs, e.g. devising models on how best to replace replicas when storage space is limited. In the context of the OptorSim work, an economic model was introduced. Most of this work had a strong focus on coupling job scheduling with data replication. These simulators introduced novel research but were not a comprehensive approach to data management, focusing only on the data placement aspect and not on the search or bookkeeping aspects.

Giggle [22] is the reference work on replica location services. It consisted of catalogues mapping logical names to physical replicas so that users could reference data by a logical name independently of its physical location. These catalogues could be layered. A s the scale increased, the authors moved to P2P-based approaches for searching data.

An example of an integrated replica management services is also very recent, by Houda Lamehamedi [19]. It consists of a P2P-based system for replica loca-

tion and an 'intelligent' framework for replication based on user demand and calculations of replication cost. This paper, despite being the most comprehensive approach 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 modeled as arbitrary file access penalties.

The SDSC SRB (Storage Resource Broker) [25] supports shared collections that can be distributed across multiple organizations and heterogeneous storage systems. It is the system that presents most similarity to our working environment but we require a system that scales further than SRB, to hundreds of sites, thousands of users and tens of petabytes of data.

2.4 Summary

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. Lustre, GFS or the Data Grid architecture). Even though a central metadata service is sufficient for most usages, such a design has limited scalability when compared with a P2P-based implementation. Another observation is that the more recent systems do not store user files as individual files on the storage. Finally, most distributed file systems maintain at most POSIX-like semantics and systems such as GFS or Hadoop are not POSIX compliant at all due to scalability issues.

Data Grids aimed from the start to support heterogeneous environments. This trend is now being adopted by distributed file systems: Hadoop already supports more than one backend. Lustre is working on a Control Panel to support bandwidth management on the WAN, enabling complex setups that spawn multiple sites ("heterogeneous" network environment).

Nonetheless, none of the existing systems matches the exact needs or environment constraints we will be addressing in our architecture. In the next section, we will look into these differences in more detail.

3 A Data Grid Architecture

Very large datasets at the terabyte or petabyte-scale often need to be hosted across multiple sites with different storage systems. Presenting a uniform, scalable data management layer is the scope of the current work.

Unlike distributed file systems that require fairly uniform setups across sites and complex network configurations, we require a management layer that can scale to hundreds of sites. Unlike commonly used P2P systems, we need to have reasonably stable associations between sites and have well established security policies. Unlike all systems presented, we need to be able to impose global policies based on data properties.

Our system needs to make opportunistic usage of volunteered resources without any centralized administration, while maintaining expected quality of ser-

vices overall. As such, we seek to combine the interesting properties of Data Grids, distributed file systems and P2P for our Data Grid Architecture.

3.1 System requirements

Even though we propose a general-purpose data management system, clearly one data handling system cannot be applicable to all possible domains. We have therefore made certain assumptions about our environment:

- For accessibility and cost reasons, data needs to be distributed among multiple computing sites rather than hosted in a single site;
- Most files are large by traditional standards, with each file being hundreds of megabytes or several gigabytes;
- Data is rarely modified after it has been produced, where the most common case is to append data rather than replace existing data;
- The production of data occurs in highly parallel environments where multiple batch nodes are producing part of a large data sample in parallel;
- There are multiple computing sites with different 'service-level agreements'. These include professionally-managed computing centers down to university clusters managed by students in their spare time. This implies very different quality of service and scale of resources;
- There are volunteered contributions of computation and data storage resources that should be supported in an opportunistic way, while taking into account their expected quality of service and size;
- There is no centralized administration of all available resources, which requires a coexistence of global and local policies;
- Volunteer contributions of resources requires the ability to adapt to different implementations: e.g. supporting different storage systems. It is expected that such need will introduce higher-failure rates and overall instability.

3.2 Core design principles

The principle design decision we took was not to depend on direct access to the servers where the files are stored. Our architecture does not replace the storage system at a site. Instead, it is layered on top of the existing storage middleware (e.g. on top of a data center-wide Lustre installation or an NFS server at a university campus). This is a completely different approach from the systems previously discussed. This considerably extends our ability to make opportunistic use of storage resources, but can lead to many more potential inconsistencies. Our design tackles these inconsistency issues. To make efficient use of the storages, we have defined an abstract layer to interact with the storage.

We decided to provide greater flexibility by not enforcing POSIX semantics, following a trend observed in other distributed systems. Users of our system require specific tools to access and manipulate data. Another important design decision is on the unit of data handling. While files are the underlying unit, all user requests are for *datasets* (groups of files). This matches our observation that

users rarely use a single file in isolation but almost always make use of groups of files (grouped statically by some semantic meaning). To further increase our flexibility and optimize storage and network usage, we have decided to decouple the units of data location from the unit of storage and the unit of transfer. Later in this paper we will look in detail into this decision.

3.3 Datasets

Datasets are natively supported by our architecture. A dataset is a collection of files, typically containing more than one physical file, which are processed together and usually comprise the input or output of a computation or data acquisition process. Datasets are always produced at a single storage system and later replicated to other storages.

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, even if other systems do not make use of it. Knowing that a dataset represents files that are used together, the system can optimize its units of data transfer and discovery. Locating datasets as opposed to files implies storing much less 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 files to replicate, it makes the most sense to replicate those files that will allow users to advance with their analysis as soon as possible - and these are typically the files part of a dataset missing at a site. Additionally, there is often the need to assign metadata attributes (e.g. 'software version used to produce the output') to a set of files. Again, in this case it makes the most sense to assign a single metadata attribute to a dataset as opposed to assign it individually to a set of files.

Creating a dataset is typically highly parallel task, where jobs in a batch system are each producing the constituent files. To facilitate the iterative process of constructing a dataset, which often lasts weeks, we have defined the possibility to create 'versions' of a dataset. Versions allow users to reference a static set of files at a moment in time. Later versions can add or remove files from the dataset. Nonetheless, for dataset integrity, datasets can only be replicated to other sites when they are 'frozen' - when no further changes are allowed. A correlation can be established with our model and the 'last close wins' semantics for distributed file system, applied to a much higher-level concept.

3.4 User Functionallity

DQ2 provides the following functionality to the users:

- A user can create a dataset. A dataset is assigned a storage at creation time. The dataset can then be modified by adding or removing files, using specific tools to handle the physical movement of data from the user's file system to the storage system. The user does not control or manage the physical location of the files within the storage system; this is done internally as we

shall describe later. The storage system is seen as a black-box from the user perspective and all interactions involve DQ2 tools.
- A user can replicate datasets between storages across the wide-area network or within a site. After a dataset has been fully defined but before it can be replicated, the user must freeze it. This guarantees the dataset can no longer change. Afterwards, the user may subscribe the dataset to another storage. The subscription, similar to the principles describe on [20], is used for asynchronous replication. There are multiple subscription options: e.g. restrict data flows by using only specific source sites (the default is for DQ2 to choose the best sources); or set the transfer priority among other options.
- A user can receive events during the replication process. As replication is the one of the primary functions of DQ2. The user can choose to receive notifications whenever certain events happen. For instance, when the dataset has been fully replicated, the system can send a notification to an endpoint specified by the user at subscription time. This is used to link the data transfer system with the job submission system: when data is available at a storage, the production management system gets a notification and automatically launches jobs to process these data. We found this mechanism to be routinely used. Subscriptions can be cancelled at any time, triggering a clean-up of any ongoing transfers.
- Users can retrieve an entire dataset or some of its files to a local file system. This allows synchronous downloading of data from the best available sources.
- Users can query DQ2 for replicas of a dataset to locate data.
- Users can also request deletion of replicas. Deletion requests are dealt with asynchronously but users are informed when querying for replicas. Similarly, a user may request the deletion of a dataset in the system: this triggers deletion of all its replicas.

3.5 Architecture

Figure 2 describes the overall architecture. To implement the functionality previously described, DQ2 uses a combination of local and global services.

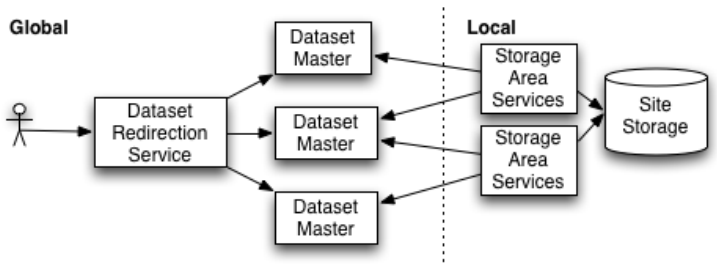


Fig. 2. DQ2 Architecture.

Local services. The local services are called *storage area services* (a storage area is loosely defined as a subset of a storage system). These local services are associated to the storage at a site and typically require privileged access, depending on whether the storage-specific plugins require such local access. There may be more than one storage area service per storage system. For DQ2, these areas are independent, each with its own set of services and dedicated disk space. Local services are designed to be minimal without any global information. This decision improves overall robustness making components more autonomous.

Storage area services have two distinct roles: to hold dataset definitions and to handle files in the storage. It is the responsibility of the storage area services where a dataset is created to hold its definition, even if other replicas are created and the master copy deleted. This information is permanent and needs to be stored in a reliable way. The other role is to physically move files to the storage from a remote location (import), to delete files from the storage, to stage files (preparing a file for export) and to lookup files (to find if a storage has a certain file). These activities are executed by local agents that rely on transient information (available only during the lifetime of the request). Coordination of which files to transfer, delete, stage or lookup is handled by a global component we describe next. Nonetheless, decisions can be overridden locally (by denying or re-ordering requests) given site or storage-specific policies.

Note that DQ2 does not include any database with full knowledge of a storage namespace. When DQ2 needs to know whether a file is present, it will query the storage system, avoiding the need to maintain and synchronize a separate database (which could cause both scalability and consistency problems).

Global services. The global services have an important role in our system. Each global service acts as the *master* for all activities with a dataset. When a user asks for a dataset to be replicated, the request is redirected to a master, using a *dataset redirection service* that guarantees unique mapping between a dataset and a master service. The master will then queue and schedule the dataset request. The master does not execute any of the activities: it simply assigns work to local agents. Work assignments include looking up and staging source files, doing wide-area transfers or deleting files. The local agents are not dataset-aware: these only deal with bulk requests of files. It is up to the *master* to optimize work assignments based on dataset knowledge.

Throughout its activity, the master will gradually build a cache of dataset (and file) replica information. This cache serves as the mechanism for users to locate dataset replicas. Dataset locations are never absolutely correct in a distributed system: it is always possible that a request fails because data was lost unexpectedly. Nonetheless, to avoid constant hits to the storage systems to locate data, we maintain a cache that is gradually renewed from the masters activity.

Quotas and accounting. In DQ2, quotas are handled in the master. DQ2 can only guarantee that within each master a user stays within his quota (or

e.g. a replication request is denied). Global accounting is possible by gathering statistics from all masters. In practice we have found the model to be sufficiently flexible: the service that assigns datasets to masters may take into account the ownership of the dataset and ensure that all datasets belonging to the same user are mapped to a single master.

4 Implementation

This section describes implementation details for DQ2. We describe the implementation of the various components and their interactions. Common across all components is the usage of HTTP as the communication protocol for client/server requests.

4.1 Dataset Redirection Service

A user specifies a dataset in every request to DQ2. The request is first sent to a central dataset redirection service. This service redirects the request, using HTTP, to the appropriate master. Our current implementation statically assigns datasets to masters based on a set of rules based on the dataset name. To avoid single points of failure, multiple instances of the service can be setup, sharing the same set of rules. Other rules could be foreseen to e.g. have the same load across all master, which would require some form of coordination among masters. In practice, this is not required from our experience and we have opted to strictly partition dataset masters.

This redirect mechanism provides partitioning of requests among multiple masters. Masters are deployed to serve a single *activity*. Examples of activities defined in our production system are *raw data taking*, when the ATLAS detector is taking raw data; *regional monte-carlo production*, encompassing all simulation activities from a regional group without interest to the collaboration; *official monte-carlo production*, including simulation activities that passed strict physics validation and hence are available for use across physics groups.

4.2 Storage Area Services

Dataset definitions. Storage area services contain dataset definitions in a relational database. At creation time, each dataset is assigned a logical name by the user and a globally unique identifier (UUID [6]) by the system. This guarantees that datasets are uniquely identified when replicated to other storages. Each file is also assigned a unique identifier by the system in addition to its logical file name.

Agents. There are different agents with distinct roles: to lookup files on the storage, to stage files (from a tape system or from the storage to an export disk buffer), transfer or delete files. Agents use in-memory structures and hold

minimal state. To interact with the storage, each agent makes use of storage-specific plugins for executing the task. For instance: the mechanism to stage files from tape depends on the tape system being used; similarly, to locate files in a storage system a *POSIX stat* command may be sufficient; in other cases, storage dependent tools are required.

Interactions with master for schedulling. Agents have a list of masters on which to poll for work. Each agent will poll for work, doing round-robin requests across its masters. For some cases, DQ2 also implements a simple *fair-share mechanism*. In this case, the agent will poll a master, specifying a maximum response size. The agent can then maintain a share allocation to each master, guaranteeing that each master gets an allocation of the agent's work (e.g. the site administrator can dedicate half its resources to serving requests from a specific master: this is regularly used in ATLAS to guarantee that raw data gets shipped in due time to all sites).

Wide-area transfers. DQ2 can support a variety of transfer protocols with its plugin approach. GridFTP is commonly used due to its broad support by storage systems but other protocols (HTTP) are also supported.

To interact with the storage systems we make use where possible of a common mass storage interface, called SRM ([8]). The SRM interface (v2.2) is implemented by several storage vendors. In some cases, direct access is still required as not all required information is exposed through SRM.

4.3 Dataset Master

Web service. The web service handles user requests to stage, transfer, delete or verify consistency of datasets. Requests may be denied immediately if quotas are exceeded. Users also contact the web service to get the status of asynchronous requests (e.g. replication status). The web service implements authentication through Grid X509 proxies ([5]). User read requests are insecure to avoid the overhead of proxy verification. All other user requests are secure.

There is a second web service endpoint used by agents to request work. The security on this web service may also be based on grid proxies but is usually configured at the firewall level, avoiding the overhead of proxy verification (reducing CPU usage on the server).

DQ2 makes use of HTTP URLs providing a simple REST [12] interface. We have found the REST interface useful for linking external systems to DQ2 (e.g. metadata catalogues refer to the dataset status by using our public dataset URLs).

Dataset-based brokering. The dataset master is responsible for assigning work to the local agents. When handling a dataset request, the master makes use of its knowledge about current replica status for decision making. Work

assignments are made synchronously as the agents ask for more work. This synchronous decision-making is an important property of our implementation, enabling a feedback approach: more work is given to a local agent as it finishes its current set of work. The work assignments are done *just in time* and thus rely on the most up to date information.

When transferring a dataset, the master will assign work giving higher priority to the datasets that have most files already present at the destination. Therefore, dataset transfers will likely last less time and the completion rate is higher. This implies that the master will scan the list of active dataset transfer to that storage, replying to the local agent with the set of files that will likely complete the most datasets in the shortest time.

Other optimizations are possible in the master. An important one is when using tape backends. The latency to recall a file from tape is usually very high. These requests can be optimized by doing the least number of tape mounts. When transferring data that is on tape at a source to another storage across the wide-area, a good coordination is required. DQ2 is able to coordinate such transfers, making bulk lookup requests at the source, segmenting stage requests per tape (based on information provided by the lookup agent) and scheduling the transfer at the destination as soon as sets of files are made available from the source. The feedback-based model and the synchronous decision making are critical properties for this mechanism.

Caches. To implement the dataset-based brokering, a very fast response to agents requests is required. As such, we have implemented various caches on the master. One cache holds contents of datasets: the first time a request comes for a dataset, the master does not know its constituent files. It must therefore ask the storage area services for the dataset definition and caches back this information. Further usages of the dataset will use this cache.

Another cache contains dataset and file replicas. DQ2 does not have ultimate knowledge of where data is located: all it can do is act based on previously known information and expect that data has not been lost in the meantime[2]. As DQ2 is notified of the state of lookup, stage and transfer request, it caches this information. Future scheduling decisions rely on this cached information (if the cache is recent) and the system will gradually renew the information as required. Users make use of this cache to locate dataset replicas (even knowing additional information, such as whether a particular file is staged, if DQ2 was required, for some other request, to stage the file).

In-memory structures. The master data structures are kept in-memory to guarantee better performance, which is important for scheduling decisions (e.g. choose files to transfer next out of the list of pending requests). In addition to

[2] In our early prototypes, we exercised mechanisms such as having a storage notify DQ2 of data losses but this approach was not possible to implement in practice due to various issues interfacing with storage systems.

in-memory structures, all data structures are written to a log on disk. This log is asynchronously fed onto a relational database. When the master process is restarted, the log is read and the state is re-initialized before the master starts serving new requests.

5 Fault Tolerance

Interactions between Master and Storage Agents. The dataset master interacts with storage area services when it needs to resolve a dataset and schedule work to a local agent. All requests are subjected to timeouts and are retried by the master. Requests to resolve a dataset will be retried indefinitely until a valid response is retrieved (as we expect to eventually have a valid response). Other requests, such as staging or transferring a file, will be attempted a maximum number of times, with an exponential back-off.

The agents also have a retrial policy when contacting the master. In case a master goes down, each agent will retry reconnecting with an exponential truncated back-off, giving the master time to recover if the service is unavailable.

Early validation of datasets. DQ2 implements early validation of user datasets. In our early prototypes, we did not explicitly validate a user dataset before attempting to transfer it. As a result, resources were being wasted trying to transfer a dataset whose data was badly uploaded, missing or lost. We found that the majority of cases corresponded to problems uploading files to the storage that were not detected.

Given that DQ2 is layered on top of existing storage systems, we decided to shield DQ2 from these errors by having a mandatory validation step before a dataset can be replicated to other sites. This step is coordinated by the master. It also serves as the mechanism for the master to be informed of the existence of the new dataset; and store it in its cache. The step is triggered automatically when the user notifies that the dataset is *frozen* (its contents are immutable).

Data corrupted or lost. In a distributed system with hundreds of storage systems, there are frequent occurrences of data corruption or data loss. The master is able, in many cases, to detect and automatically correct these occurrences. When files cannot be repeatedly accessed, the master requests the storage with suspected data to copy over the files again from another available source. At the same time, it blacklists those replicas so that other interested parties avoid them. The mechanism is efficient given that the master possesses global knowledge of the file replicas for a dataset.

Master availability. The master relies on in-memory structures. Operations on the master are checkpointed to disk. There is an asynchronous system feeding the checkpoint log onto a relational database for increased redundancy. Additional redundancy mechanisms are possible, such as having master/slave replication of

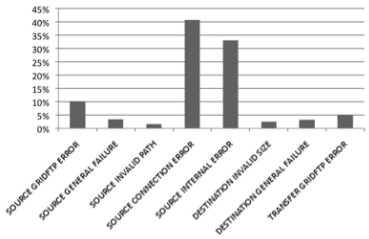


Fig. 3. Dominant errors classes (over a 1-month period).

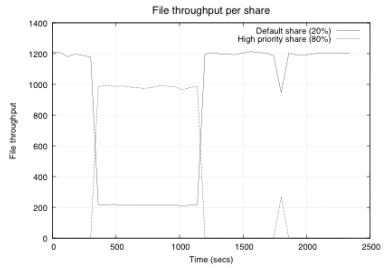
the database holding the log information or splitting the web server from the process executing the requests.

Transfer reliability. Figure 3 shows failure rates observed in our production instance of DQ2. The majority of errors are reading data from storage systems. For this reason, DQ2 validates source files prior to transfer, by doing a source storage lookup. Only when files are reported as *found* and *staged* at the source can the transfer start.

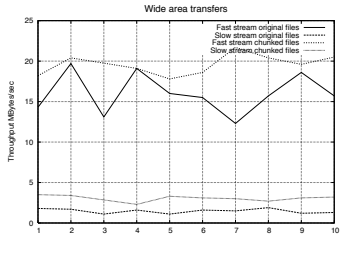
For wide-area transfers, DQ2 implements a retrial strategy that takes into account previous transfer history and *channel* performance. A channel is a virtual unidirectional link[3] between a source and destination storages. Best performing channels will serve more transfer requests, given the feedback-based model (polling for more work when work is done) implemented between the agents and master. Therefore, given more than one possible source for a file, it is likely that the channel performing the best will serve the file first. If a transfer between a source and destination is persistently failing, the agent responsible for collecting work at the destination side will back-off and not request more work for some time. If a specific file transfer is permanently failing, the master will temporarily blacklist the source, allowing other sources to be used. If the failure is very frequent for a single file, the file is marked as corrupted and there is an attempt to copy it over from another location.

To validate transfers, we choose ADLER32 checksum because of the its rolling hash property, which allows the checksum to be computed as the input moves through a window. This eases the checksum computation without introducing significant overheads (e.g. such as having to re-read the file to compute the checksum). Tape drives also often compute ADLER32 at the hardware level when writing files to tape, which provides another verification step in a optimized manner.

[3] e.g. 'CERN to BNL' is the channel serving requests from CERN to the Brookhaven National Laboratory in the US.



(a) Distribution of files copied per share.



(b) Comparing usage of export buffers with original and chunked files.

Fig. 4. Overview of scalability properties of DQ2.

6 System Scalability and Data Availability

In this section we describe the mechanisms in DQ2 that guarantee high availability of data and scalability of the system. We also describe the scale on which DQ2 is operating on a daily basis for the ATLAS Experiment, moving petabytes of data every month.

Separation between global and local services. The separation between *storage area services* and *dataset masters* has multiple advantages. One is avoiding having global knowledge present on the local services. Another advantage is partitioning the system for scalability while maintaining global knowledge about replicas of a dataset. This knowledge allows the master to make better decisions about how to handle dataset requests, as it knows the state of the various replicas (if on disk, if being garbage collected, etc). The storage agents continue to have the ability to throttle access to their storages by simply not asking one of the masters for more work.

Tracing dataset popularity. As the number of datasets increase in the system, we expect to have older datasets become less interesting with time. These are usually kept for archival only (often on tape storage) but are no longer regularly used. DQ2 includes a *tracer service* that records all usages of a dataset. This is used for monitoring purposes but could also be used for internal optimizations of the system. Thus, we can detect which datasets are more popular as to predict hot-spots and implement automatic replication. Similarly, if the master is unable to keep with the number of datasets it needs to manage, dataset usage information could be used to rebalance the load, ignoring unused datasets.

Competition between transfers. When transferring datasets between sites, the local agent needs to decide between competing requests, as it is needs to serve

multiple masters. Fair-sharing is used to guarantee fair split of resource usages. Figure 4(a) from a simulation run illustrates the results of our algorithm with high priority transfers taking over the channel as needed and according to shares. Each of these shares maps to a different master, serving different datasets. There are obvious limitations with this model, such as assuming that all file transfers are equal within a channel (regardless of file sizes). This is being addressed in newer versions but has served us well in practice.

Improving data availability with import/export buffers. After a file is staged at the source but before being transferred to a remote storage, DQ2 can optionally copy it to a export buffer managed by DQ2. Similarly, when importing data, the destination storage may first place the file onto an import buffer before writing it to its final location.

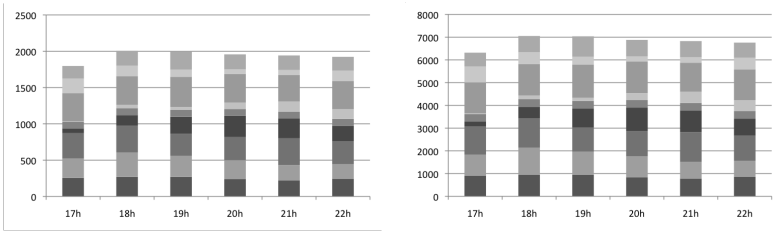
These buffers allow DQ2 to split the units of storage from the unit of transfer: the file may be artificially split or merged for transfer and/or storage. This can be used to improve storage and transfers and protect DQ2 from storage instabilities.

If a storage has a tape-backend there is a high cost in the mechanical process of mounting a tape for reading back the data. If a dataset is sufficiently large but its constituents are small files, it is convenient to aggregate files of a dataset into larger units as to improve later recalls from tape. As in DQ2 a dataset is usually read in its entirety, this leads to increase performance and avoids clustering datasets between different tapes.

Figure 4(b) also illustrates improvements achieved in throughput with this technique. The figure shows results from (HTTP-based) wide-area transfers where we compare simple HTTP file transfers with a mode where each file is split into smaller chunks. In these tests, 64 MByte chunks were used. When transfers are chunked, a slow read of a big file from a server has less performance impact on concurrent transfers, because each HTTP request has a shorter lifetime as it transmits less data. Transmitting long files in a single request blocks a server 'slot' for a long time, affecting parallel transfers. Our tests were conducted on loaded servers (10 to 15 clients) with samples of real ATLAS data. The 'slow stream' clients were artificially slowed down to match typical competition patterns we observe in our production system, where transfer rates to destination storages with good connectivity get affected by concurrent transfers from the same servers to destinations with bad connectivity.

Real World Usage. The ATLAS production instance of DQ2 currently hosts over *1.6 Million* datasets. There are over *50 Million unique files* with a total of *80 Million replicas* (the average replication factor for ATLAS data is relatively small due to lack of disk space). The system is now hosting *˜7.4 PetaBytes* of data over *60 distinct computing centers*. One observation is the scale difference between number of datasets and files, which motivates our choice for natively supporting datasets.

Figure 5 shows results of large-scale transfer tests using DQ2. In these tests, we transferred datasets from CERN to our major data centers. The figure covers



(a) Aggregate throughput in MBytes/sec.

(b) Data copied in GigaBytes.

Fig. 5. Overview of large-scale tests with DQ2 to multiple computing centers during 6-hour period.

a large-scale test where the system maintains an average throughput of over 1.5 Gigabytes/s. During this period, storage areas went down and came back online later, showing the resilience of the system to the frequent occurrence of temporary failures. The system achieved the rate of 7 TeraBytes of data exported per hour.

7 Future Work and Conclusion

In this paper we addressed the problem of managing very large datasets in a distributed environment. After presenting and discussing the state-of-the-art as well as recent trends, we introduced a new system developed using the results of previous research on P2P, Data Grids and distributed file systems. Our major contribution is the provision of a more comprehensive feature set for managing very large distributed datasets in a heterogeneous environment. DQ2 is managing over 7 PetaBytes of data and has achieved transfer throughput in excess of 1.5 Gigabytes/s. Future work will focus on increasing the scalability of DQ2 and protecting the system from lower-level middleware instabilities. We will also conduct dedicated reliability tests to demonstrate the robustness of the system.

Acknowledgments. We would like to acknowledge the many contributions to the design by Torre Wenaus and David Cameron and the help of David and Benjamin Gaidioz in implementing DQ2.

References

1. The ATLAS Collaboration, `http://atlasexperiment.org/` (1999)
2. A. Chervenak et al., "The Data Grid: Towards an architecture for the distributed management and analysis of large scientific datasets," J. Network and Comp. App., Vol. 23, 187-200 (2001)
3. W. H. Bell et al., "Simulation of dynamic grid replication strategies in OptorSim," in Proc. Grid Computing - GRID 2002 : 3rd Int. Workshop, USA (2002)

4. J. Risson et al., "Survey of research towards robust peer-to-peer networks: search methods," in Computer Networks, Vol. 50, Iss. 17 (2006)
5. I. Foster et al., "A security architecture for computational grids," in CCS 98: Proc. of the 5th ACM conference on Computer and communications security, ACM Press, NY, USA, 83-92 (1998)
6. International Standard "Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 Object Identifier components" (ITU-T Rec. X.667 — ISO/IEC 9834-8)
7. W. Allcock et al., "GridFTP protocol specification," Technical report, GGF GridFTP WG (2002)
8. A. Shoshani et al. "Storage resource managers: Middleware components for grid storage," in Proc. of Nineteenth IEEE Symposium on Mass Storage Systems (2002)
9. J. H. Howard et al., "Scale and performance in a distributed file system," ACM Trans. Comput. Syst., Vol. 6, No. 1, pp. 51-81 (1988)
10. R. Sandberg et al., "Design and implementation of the Sun Network Filesystem," in Proc. of the Summer 1985 USENIX Conference, pp. 119130, Portland, OR (USA) (1985)
11. S. Ghemawat et al., "The Google File System", 19th ACM Symp. on Op. Sys. Princ., NY (2003)
12. R. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. Thesis, University of California (2000)
13. R. Rocha et al., "Monitoring the ATLAS Distributed Data Management System," in Proc. of Computing in High Energy and Nuclear Physics (CHEP) (2007)
14. E. Cohen et al., "Replication Strategies in Unstructured Peer-to-Peer Networks," in Proc. of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications, USA (2002)
15. M. Satyanarayanan et al.. "Coda: a highly available file system for a distributed workstation environment," in IEEE Trans. on Comp., Vol. 39, No. 4 (1990)
16. F. Schmuck et al., "GPFS: A Shared-Disk File System for Large Computing Clusters," in Proc. of the 1st USENIX Conference on File and Storage Technologies, (2002)
17. P. Andrews et al., "Massive High-Performance Global File Systems for Grid Computing," in IEEE Conference on High Perf. Net. and Comp. (2005)
18. J. Bester et al., "GASS: a data movement and access service for wide area computing systems," in Proc. of the 6th workshop on I/O in parallel and dist. systems (1999)
19. H. Lamehamedi et al.. "Data replication strategies in grid environments," in Algorithms and Architectures for Parallel Processing (2002)
20. A. Samar et al., "Grid Data Management Pilot (GDMP): A Tool for Wide Area Replication," in IASTED International Conference on Applied Informatics (2001)
21. P. Schwan, "Lustre: Building a file system for 1000-node clusters", in Proc. of the 2003 Linux Symposium (2003)
22. A. Chervenak et al., "Giggle: a framework for constructing scalable replica location services", in SC2002, Baltimore, USA (2002)
23. Q. Liv et al., "Search and Replication in Unstructured Peer-to-Peer Networks," in Proc. of the 16th international conference on Supercomputing, NY, USA (2002)
24. P. Kunszt et al., "Data storage, access and catalogs in gLite," Local to Global Data Interoperability - Challenges and Technologies, pp. 166-170 (2005)
25. C. Baru et al., "The SDSC storage resource broker", in Proc. of the 1998 conference of the Centre for Advanced Studies on Collaborative research, pp. 5 (1998)