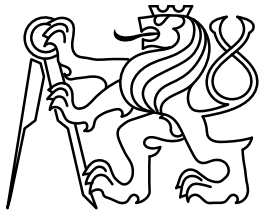
CZECH TECHNICAL UNIVERSITY AT PRAGUE

Faculty of Nuclear Sciences and Physical Engineering

Department of Mathematics



MANAGING WIDELY DISTRIBUTED DATA-SETS

(From physical to logical file)

Research report

Author: Pavel Jakl

Supervisor: Dr. Jérôme Lauret, Brookhaven National Laboratory, USA

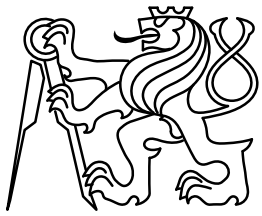
Consultant: Dr. Michal Šumbera, Nuclear Physics Institute AS CR

School year: 2005/2006

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta jaderná a fyzikálně inženýrská

Katedra matematiky



DISTRIBUOVANÁ SPRÁVA KOLEKCÍ DAT

(Přechod od fyzického k logickému názvu souboru)

Výzkumný úkol

Vypracoval: Pavel Jakl

Vedoucí práce: Dr. Jérôme Lauret, Brookhaven National Laboratory, USA

Konzultant: Dr. Michal Šumbera, Ústav jaderné fyziky, Akademie věd ČR

Školní rok: 2005/2006

Prohlašuji, že jsem tento výzkumný úkol napsal samostatně s použitím pouze citované literatury.

V Praze dne 15. září 2006

Pavel Jakl

Contents

Chapter 1

Introduction

Driven by increasingly complex problems and propelled by increasingly powerful technology, today's science is as much based on computation as it is on collaboration and efforts of individual experimentalists and theorists. But even as computer power, data storage, and communication continue to improve exponentially, computational resources are failing to keep up with what scientists demand of them. Their requirements are several times bigger than it is possible to offer under usual and available conditions.

Whether mapping the human genome, imaging the earth's substructure in search of new energy reserves, or discovering new subatomic particles, data-intensive applications are placing intensive pressure on enterprise computing and storage environments. To be more specific, thirty years ago, the high-energy and nuclear physicists were happy to create *bubble chamber* photographs providing the aesthetically most appealing visualization of sub-nuclear collisions. Nowadays, they do not only collect several order of magnitude more particles but they also want to measure particle energies, the behavior of the collisions and their following directions in real time as well as the correlation amongst the different particle produced and this, with very high accuracy.

This involves collecting a large amount of data (magnitude of several *Peta-Bytes (PBs)*) which greatly overcome today's personal computers shipped with up to 300 gigabytes (GB) of storage space and requires advanced data mining techniques as well as vast resources. Hence already in 1960's, the idea of building high performance computing clusters came to birth, and we sometimes associate the the emergence of this idea (at least as soon as they couldn't fit all their work into one computer) with the invention of "*clusters*".

In general, the aim of the high performance computing clusters is to provide large amounts of aggregate computing power for one or more users (applications). High performance computing started first with dominanace of the highly specialized *supercomputers* which are computing systems comprised of multiple (usually mass-produced) processors linked together in a single system.

In recent years, these computers have largely been replaced by lower cost *Linux cluster* computing configurations which further led to idea of building powerful Grids associating many clusters, even separated computers connected by a network around the world. Grids provides the ability to perform computations on large data sets, by breaking them down into many smaller ones, or provide the ability to perform many more computations at once than would not be possible on a single computer. The purpose is to build a single uniform access to big computing power and solve large-scale computation and storage problems.

Whether one refers to the grid as just a pure *Computational grid* or in advance a *Data grid*, one needs to deal with the reservation and co-scheduling of storage resources, similar to reservation and scheduling of compute resources. Unfortunately, storage architectures have not kept pace with growing size of computer clusters being widely distributed. Distributed means that storage is spread over N computers (N is

size of cluster) or the storage is being accessed by many physically distributed and simultaneous requests.

There are several problems and challenges that such storage systems must solve. It needs to face issues of name-space, safe keeping of meta-data, data replication access and coherence to cite only those. It needs to provide stable performance in data access with consideration to scalable potential. Last but not least, the storage system needs to provide robustness in the case of infrastructure failures and recover automatically from failures. This is especially important in distributed environment where "*remote troubleshooting*" is from problematic perspective not possible.

In this work, we will present some of many requirements with constitute a robust data storage solution and will do so by putting into perspective different storage architectures, topologies and approaches. We will first introduce several current available solutions, make theoretical comparison between them and show their architectures as well as their drawbacks. We will then focus on highlight one solution, explain the rationales behind its choice, shows its performance and scalability as results of the evaluation in the real environment of the High Energy and Nuclear Physics experiment.

For the future work, we will present the plans to improve this solution, in order to fully satisfy experiment's needs and requirements within a fully distributed computing environement.

Chapter 2

A case study

2.1 STAR experiment

STAR experiment [1] is one of the four physics experiments at the Relativistic Heavy Ion Collider (RHIC) [2] at Brookhaven National Laboratory (BNL) [3] on Long Island, New York. The Solenoidal Tracker (STAR) is the detector located at the 6 o'clock position at the RHIC collider ring.

The primary goal of this field of research is to re-create in the laboratory a new state of matter, the quark-gluon plasma (QGP) [4], which is predicted by the standard model of particle physics to have existed ten millionths of a second after the Big Bang (origin of the Universe) and may exist in the cores of very dense stars. It is used to search for signatures of quark-gluon plasma formation and investigates the behavior of strongly interacting matter at high energy density by focusing on measurements of hadron production over a large solid angle. STAR measure many observables simultaneously to study signatures of a possible QGP phase transition and the space-time evolution of the collision process.

STAR collaboration consists of more than 500 physicists and scientists from 52 institutions in 12 countries around the world. As several physics topics in the field of Nuclear Physics is limited by the available statistics for a measurement, the more statistics one may have and, the more accurate measurement is. Hence, the experiment has generated enormous amount of data since its start at May/June 2000. The word enormous means data-sets magnitude of several Peta bytes (10^{15} of byte) of data stored in over 10 millions of files over past 6 years of collider's running. The picture

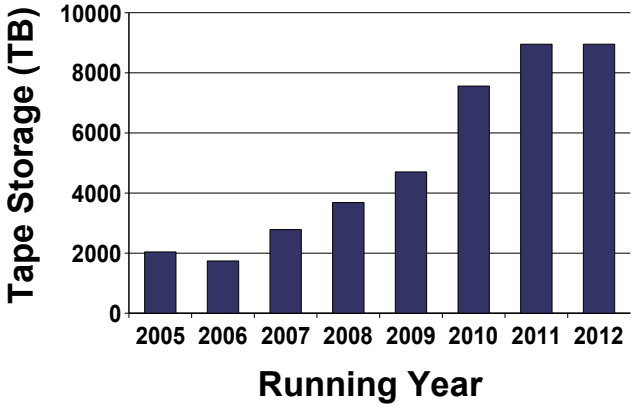


Figure 2.1: Raw data projection for the STAR experiment. Since one data mining pass leads to an equal amount of Physics data, these numbers need to be multiplied by a factor of 2 to reflect the the total Raw+Physics usable

2.1 shows size expectations of data for the up-coming years with regard of STAR's physics program. Those projections take into account planned upgrades which will generate increase of data to be taken as needed by evolution of the Physics program. This huge amount of data certainly brings a need of large computing environment and infrastructure which allows firstly reconstruct and further analyze data by many physicists interested in different kinds of STAR physics observables. A significant part of the computing effort has been focused on developing software for management and

maintenance of storage, mainly developed in PERL language [5], [6], [7] and using MySQL database [8], [9], the model is described later in the section 2.5.

2.2 STAR computing resources

STAR utilizes primarily two computing facilities, the RHIC Computing Facility (RCF) located at the Brookhaven National Laboratory(BNL), and the Parallel Distributed System Facility(PDSF), a sub-cluster complex of the National Energy Research Scientific Computing Center(NERSC) at Lawrence Berkeley National Laboratory (LBNL). Both of these facilities are Linux-based cluster with about thousand of CPUs, several hundreds TBs of disk space and PBs of the tape space.

The RCF/BNL facility is the location where the work described in this report took place. The RHIC Computing facility disposes with range from having dual Pentium III 450 MHz processors to dual 3.2 GHz Pentium IV processors. They have between 512 MB - 2 GB RAM, and 40 GB - 400 GB local disk storage. The older machines have 100 Mb network interfaces, but the newer, faster machines have Gb Ethernet network interfaces. These machines were originally configured with the RedHat Linux distribution, but have migrated to Scientific Linux, which is RedHat Linux open source code, compiled and distributed by Fermi National Accelerator Laboratory for better achievement and support of physics needs and goals.

Data work-flow from reconstruction to analysis

When the collider is running, the storage of raw data taken directly by each subsystem of the STAR detector is accomplished using a hierarchical system of IBM servers, StorageTek tape silos, and HPSS (High Performance Storage System) software.
After the raw data is archived, a reconstruction process is started: data are moved from tape to disk so that measurements gathered by different sensors can be sorted by time and packaged into individual events. The reconstruction process is each time individual and dependent on currently running physics program. Although, the process has identical characteristics that must move the raw data from tape into local disk, pass all files with the help of batch system to distribute and control the processing into many independent jobs.
Each job moves results after accomplishment from a local disk into the permanent space which it is again the tape system (HPSS). The tape system offers cheap and reliable storage, but on opposite side very slow access and data needs to be usually migrated on temporary and faster locations where could be available for physicist's analysis. These temporary locations for fast access involve a building of sophisticated large storage architectures and solutions. This topic would be discussed in following sections.

2.3 Storage solutions in STAR: Centralized vs distributed

There are three coexistent methods mentioned in [10], [11] for connecting disk-based storage to computing nodes for fast access. These three methods could be merged and re-grouped into two main topologies/approaches:

- **Centralized storage** - a centralized storage is a storage with many heterogeneous servers connected to one single storage space. The single storage space can have heterogeneous storage entities or disk drives. With centralized storage solution, there two mainstream sub-groups corresponding to two main architecture choice or strategies:
  - ◇ *Network Attached Storage (NAS)* - NAS systems usually contain one or more hard disks, often arranged into logical, redundant storage containers or RAID arrays connected over the network to computing node. The containers are called NAS *filers*.
  - ◇ *Storage Area Networks (SAN)* - SAN is dedicated, high performance storage network that transfers data between servers and storage devices, separate from the local area network connected to computing nodes using **Fibre Channel**. The Fibre Channel is a high performance network technology designed to bring speed and flexibility to multiple disc drive storage systems.
- **Distributed storage** - a distributed storage is a storage that has many geographically-dispersed disk drive units, usually spread over many hosts or servers. All the hosts or servers are connected together through the network.
  - ◇ *Direct Attached Storage (DAS)* - DAS is the most basic level of storage in which storage devices are part of the host computer and directly attached to it. The computing nodes must therefore "*physically*" contact the server (the host owning the storage) in order to connect to the storage device. Unix systems implements the NFS (Network File System) protocol as on way to communicate between server and clients in a uniform manner. Most NFS architecture suffers from the single point of failure represented by the server node.

Choosing the right storage solution can be a very difficult question, because there is no right answer for everyone. It is very important to look he load and usage profile but also the long-term plans and requirements of the current organization. Several key criteria which could be considered include:

- **Performance** - aggregate I/O and throughput requirements of the system
- **Capacity** - the amount of data to be stored
- **Scalability** - possible long-term and easy growth of the storage system
- **Availability and Reliability** - storage is on-line 24/7 without any disruptions

- **Data protection** - recovery and/or backup requirements
- **Maintenance** - human resources requirements and cost for maintenance of the system
- **Budget concerns** - initial purchase price in regards of the storage volume

Since the whole discussion is dedicated to having very high performance, reliable and fast access to physics data one could imagine that well-known SAN with its "Fibre Channel" could be the best choice. However, there are many circumstances when deciding between NAS and SAN is not an easy task.

Firstly, SAN solutions come into flavors much faster than what was available 5 years ago. During the past five years the transfer rate for leading edge Fibre Channel has increased fivefold from 20MB per second to 100MB per second per one fiber. Over this same period, however, the transfer rate for leading edge networking interconnects has increased tenfold from copper based connection at 12.5MB per second for 100baseT Ethernet to 128MB per second for Gigabit Ethernet per machine or client. In other words, network data rates before possible from solutions reserved for high end servers are now so inexpensive that they have become commodities, making possible an aggregate IO (over many clients) which cannot be absorbed by a single fiber channel NAS.
Secondly, NAS is easier to understand than SAN. SAN are very complex with its infrastructure; one has to firstly understand Fibre Channels, then the switch manual, and the manuals that come with any SAN management software. The concepts of Fibre Channel, arbitrated loop, fabric lo-gin, and device virtualization are not always easy to grasp.
Thirdly, NAS is easier to maintain than SAN. SANs are composed of pieces of hardware from potentially many vendors. SANs have therefore a larger number of components that can fail and fewer tools to troubleshoot these failures, and more possibilities of finger pointing.
Lastly, NAS is much cheaper than SAN. Again, since NAS let you leverage your existing network infrastructure, they are usually much cheaper to implement than a SAN.

Although, it may seem that NAS is a more appropriate solution for each environment, there could be instances where the SAN's aggregate throughput can overbalance its cost and complexity.
Another dimension of the question comes when one start considering centralized storage versus distributed. The question "**DAS, NAS, SAN ?** " could perhaps be reduced just to "**Centralized vs distributed ?**" as we will explain.

Although distributed storage introduce many components within a complex server/server and server/clients layout, from economical statistics, the initial purchase price is cheaper by factor of 10 comparing to the distributed storage. As a consequence, even though the implementation of centralized storage is growing at a faster rate than that of distributed storage (mainly due to the lack of ready-to-use solution to manage data distribution), its cost cannot compete to the possibility

offered by distributed storage solutions.

When considering distributed disk, it is important to understand what the data availability requirements are. In order for clients on the network to access the storage device, they must be able to access the server it is connected to, speaking nothing of getting information which server contact. If the server is down or experiencing problems, it will have a direct impact on user's ability to access the data. In addition, the server also bears the load of processing applications which can at the end slow the IO throughput as we mentioned in a previous section.

When applying this discussion to studied case of having fast access to physics data and making a conclusion, the availability and data protection aspects in distributed storage could be reduced by re-copying the lost data from master copy on the tape drive to the other server. Speaking about scalability and capacity of distributed storage, one could imagine linear growth of storage simultaneously with computing nodes, since the storage is attached. There is no other need for extra hardware in order to increase the size of the storage. The maintenance resources are reduced in case of distributed disk, since there is no need of having two separated persons for maintaining computing and storage element, one person can serve both of them.

As a conclusion, the distributed storage seems as a better solution for physics data and is bringing cheaper, scalable, capable solution, but on the other hand worse manageability, sometimes called: "Islands of information". The difficulty relies on management of space spread among multiple servers, not mentioning load balancing issue, obtaining highest performance and scalability (since CPU and storage are now coexisting).

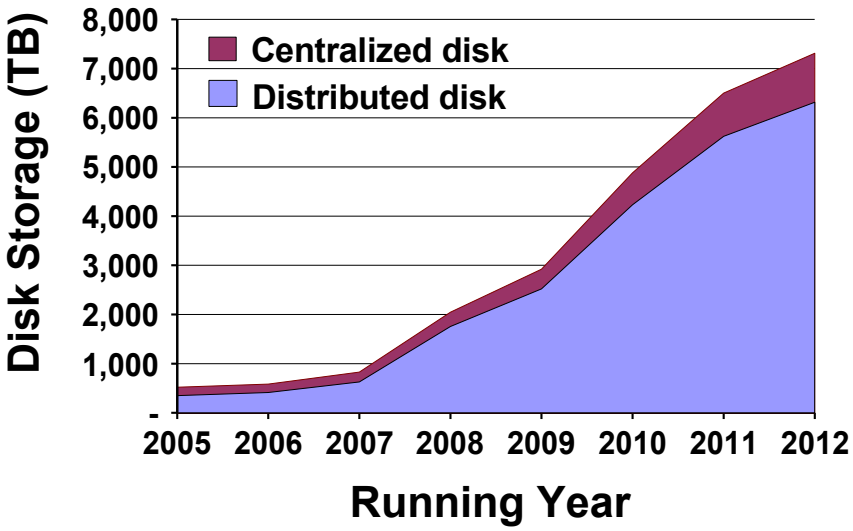


Figure 2.2: RHIC computing facility capacity profile

Driven by the need for vast amount of data and economics, the STAR software & computing project has taken the decision to move toward to a distributed storage

model infrastructure as their primary storage solution as illustrated in Figure 2.2. This satisfies the needs of the collaboration and the requirements for the upcoming years. To be more concrete, STAR now-days disposes, at the RCF with 150 TB of disk space spread over 400 nodes and 75 TB of centralized storage implemented by SAN and exposed to the users via NFS. This is not without challenge and the next chapters will be focused on the architectural, methodological, algorithmic and technological aspects of the publication, management and performing large scale data access on the distributed storage.

2.4 Hardware vs Software solution

By this time, we have already discussed centralized and distributed storage topologies with their benefits/handicaps from viewpoint of infrastructure issues trying to address main goals and postulates of the ability to somehow publish data. The previous question of "Centralized vs distributed ?" should be discuss from different vision as: "Hardware vs software solution" with reference to current nowadays implementations.

At this time, current centralized file system designs fundamentally limit performance and availability since all read misses and all disk writes go through central server or head node.To increase performance, reliability, availability and reduce possible bottlenecks, a typical installation relies on specialized server machines/hardware configured with multiple processors, I/O channels, and I/O processors or even using pure hardware architectures such as RAID.

In order to address reliability problems of failing devices as well as to improve the I/O performance in centralized solutions, the concept of RAID (Redundant Array of Inexpensive/Independent Disks) [12] has been widely adopted over the past years. The cited publication originally detailed five strategies, usually referred to as RAID levels and numbered from 1 to 5, which use different strategies to protect against data loss, to improve the performance, or both. Later on, additional and hybrid RAID levels have been developed, among them linear RAID, RAID-0, RAID-6, RAID-10, or RAID-53, to name the most common ones. The basic idea of all RAIDs is to increase data availability to enhance reliability and/or performance. One strategy used by several RAID levels is striping: logically consecutive data is subdivided into blocks, which are stored in a round-robin fashion. Second strategy used is: a parity block - the exclusive OR of the corresponding bits of data blocks being striped.

This improves the transfer rate in streaming mode as multiple requests can be issued to the constituent devices in parallel fashion. The actual RAID level determines, how redundant the information is generated, if at all, and how the data and the redundancy information is spread over the devices.
Linear RAID is a simple concatenation of several disks into one large virtual device. It provides no protection against device failures and the performance does not differ from that of a single disk. RAID-0 uses all its constituent devices for data striping. As in linear RAID, no redundancy algorithm protects from data loss. However, due to the application of striping it provides better transfer rates. RAID-1 duplicates

data to a clone disk. Hence, in case of a disk failure, the data is still available by means of the other disk. Read accesses can also be sped up by reading the data in a stripe fashion. RAID-2 uses Hamming codes, which extends parity check and bit errors.At a comparable performance it needs more space than RAID-3 and is hence rarely implemented. The latter RAID level stripes data on a byte level. As all disks are accessed in every I/O operation, RAID-3 delivers a high transfer rate, but works poorly for applications with high request rates. Data protection is achieved by parity storage on a dedicated device. RAID-4 and RAID-5 stripe on block level and access each disk individually. While RAID-4 uses a dedicated parity device, which can easily become a bottleneck, RAID-5 distributes the parity information over all underlying devices for load balancing purposes. As RAID-5 is a compromise between performance and reliability, it is very widely used in most centralized solution. A more detailed discussion of the advantages and shortcomings of the different RAID levels can be found in [13].

While RAID offers performance and reliability, it suffers from 2 limitations. First, the overhead of parity management in all RAID levels can hurt performance for small writes; if the system does not simultaneously overwrite all N-1 blocks of a stripe, it must first read the old parity and some of the old data from the disks to compute the new parity. A second and most significant drawback of commercially available hardware RAID systems is that they are significantly more expensive that non-RAID commodity disks because they need to have special-purpose hardware to compute parity.

No matter, if the centralized solution is using architecture such as RAID and its advance striping, a central server represents a single point of failure requiring server replication for high availability. Replication surely increases the cost and complexity of central servers and can increase latency on writes since the system must replicate data at multiple servers. The overhead can even grow and be functionally dependent on the size of the repository.

One of the file system and storage solution, trying to solve a problem having one central server as a bottleneck is **Panasas** file system and storage cluster [14], [15]. The core of the Panasas architecture is a cluster of intelligent "*StorageBlades*". These hardware-based devices provides the storage and parallel transfer capabilities of the system. They are pooled into a cluster that provides the capacity and load balancing across the cluster, and fault tolerance through the RAID reconstruction. Additional cluster of "*DirectorBlades*" forms the meta-data management layer and acts as a protocol gateway to support other file system and data management protocols, including NFS and CIFS.

The other file system worth to be mentioned is **Lustre** [16], which aims to provide a scalable, high performance file system in more software fashion way than Panasas. Its architecture follows the client-server paradigm using multiple servers, the so-called I/O daemons. These daemons usually run on I/O nodes, which are special nodes in the cluster dedicated to I/O. User applications run on compute nodes, which need not to be distinct from I/O nodes. A meta-data manager handles

all meta-data information. This manager is contacted by user processes for operations such as open, close, create, or remove. The meta-data manager returns information that is used by the clients to contact the I/O daemons for direct file access. The meta-data manager is contacted in each case of file system meta-data changes. In Lustre's terminology, the servers are called Object Storage Targets (OSTs). They manage the data stored on the underlying devices, the so-called Object Based Disks (OBDs). This two-fold storage abstraction allows for an easy integration of smart storage devices with object-oriented allocation and data management in hardware and follows the general trend of offloading I/O to the devices itself.
Even if the file system is somehow distributed on dedicated server, there is still the question whether the central meta-data server could become a bottleneck whenever the cluster grows.

The high-performance storage solution, known as the **General Parallel file system** (GPFS) [17] coming from IBM try to address the meta-data issue Lustre doesn't consider. In contrast with other file systems, GPFS does not require a central meta-data server. Instead, meta-data is handled at the node which is using the file. This approach avoids the central server becoming a bottleneck in meta-data intensive applications, and it also eliminates this server as a possible single point of failure. Having the handling of meta-data at the level of I/O operation without any more complex organization and infrastructure has been already proved by GPFS as limiting the number of users and nodes serving data. While resolving one issue, another comes into the picture: the synchronization of requests and meta-data handling.

The **Google file system** [18] is an example of distributed storage system taking a dramatic different approach: it makes use of the knowledge of the application needs by moving the data closer to the application during system operation. After processes migration the data is accessible on the local node, reducing both the CPU load and the load on the network. Another drawback common to almost all storage systems is the neglect of sophisticated reliability mechanisms. The preferred approach to protect against data loss - if the system provides such a protection at all - is by generating replicas, which reduces the amount of usable storage capacity significantly but increase availability. Another approach is having the primary copy of a file at any kind of cheap storage solution such as a tape and in case of need, such data can be quickly migrated back to live storage.

A summarizing comparison of all mentioned architectures and file system is difficult and maybe inappropriate, since they differ in many aspects, such as interfaces, hardware requirements, abstraction levels, reliability, and performance (in terms of latency or throughput).
However, a characteristic common to almost all these systems is that they do not take into account the possible benefit of adapting the architecture to the characteristics of a broad range of applications, i.e. to their access behavior and their inherent independence of tasks. In many applications read accesses tend to occur much more frequently than write accesses. Most data is written once but read multiple times. This is applicable to all kinds of applications performing searches, data analysis, data mining, data retrieval, and information extraction. In addition, many

applications can be split into several independent tasks processing independent data. One examples of such application is already presented High energy and nuclear application searches for rare particles and tracks in many files.

A common denominator is also that the performance bottleneck of these systems is usually meta-data handling inside the file systems, since meta-data operations could make up over 50% of all filesystem operations [19]. These solutions usually do not scale well with increased number of clients and data, even with using a specialized expensive hardware.
The answer to the initial question of hardware vs software solutions, there is innumerable number of variants and combination hardware and software solutions, some-when with higher weight on hardware and some-while on software. Software solutions involve cheaper price and portability of the system, but on the other side possible lacking performance introduced by special hardware. As defense of the software solutions, hardware solution has an additional drawback of relying on software and drivers for hardware not being portable on all platforms. Their obsolete risk is therefore high/price indeed advantages. Moreover, with architectural, methodological, algorithmic, technological aspects and experiences can possible achieve same results seen with hardware improved solutions.

2.5 STAR data computing model

This section outlines STAR data computing model [20], introducing main components and architecture of the model. The following discussion wants gives a more precise description of the system architecture, its performance, and scalability characteristics of the system, which was used at STAR for the past five years. The characterization of the model wants to also give an overview on drawbacks and issues, not seen directly during the architectural process. I accomplished this objective by keeping the description of the whole system at low and informative level, and omitted too technical and implementation details.

One can simply imagine a naive approach of using NFS-like solution for managing the distributed storage. However, this is limited by the ability of the infrastructure and software to efficiently balance the load amongst the data servers (a few) available to the users or the data manager. Moreover, a deeper requirements analysis shows that this kind of solution is not acceptable for many well-known reasons. Thousands of concurrent accesses from end users batch jobs that continuously analyze the data in a completely random way would for example greatly overcome the scalability of the basic NFS architecture. In addition to the scalability, the NFS configuration grows exponentially with each added node. To achieve load balancing in such environment, one could imagine spreading the data-set randomly on all available storage but this would imply a simple yet additional complex organization of the data and cataloging capabilities. But even though one reverts to such techniques, NFS would still expose large pools of disk subject to equally massive data losses or corruption on crashes or hardware failures. User's applications are usually not much tolerant to such events and if a file simply does not exists, it is likely to see the application dies without any

possibility to search for other "copy" of a file.

To overcome some of these limitations, the natural alternative is building a "mesh" of loosely coupled data servers interconnected by a communication network. **ROOTD** [21] provides a remote file access mechanism via TCP/IP-based data server daemon within the ROOT framework [22]. ROOT is an object data-analysis framework used by HENP experiments. One of the most important characteristics of the ROOT framework is its I/O structure, in which every data object belonging to the framework can be streamed to disk and appended to an internally structured file. A ROOT client data access method then provides a way to reach the remote data transparently to the users of the framework.

Any experiment facing Peta bytes scale problems are in need for a highly scalable hierarchical storage system to keep a permanent copy of the data. STAR uses a High Performance Storage System called HPSS [23]. Having a large archive is not sufficient of course as million of files would make the recovery of one file a needle in a hay stack nightmare. The second vital component is to arm the experiment with a robust and scalable catalog, keeping the millions of files and potentially, an order of magnitude higher number of file replicas at reach (i.e where the data are located). If we started to speak about the cataloging requirements, we need to define conceptions and differences between file, replica and meta-data of a file.

From a basic perspective, all data accesses systems or file systems must provide a way to store the information or content of a "**file**" and a way to retrieve that data. In its simplest form, a file stores a piece of information. A piece of information can be a bit of text (e.g., a letter, program source code, etc.), a graphic image, a database, or any collection of bytes a user wishes to store permanently. The size of data stored may range from only a few bytes to the maximum size of the file limited by the particular file system. For a structure imagination, a file could be presented and expounded as a *"stream of bytes"*.
A file system also stores separately information related to the "*file*"; these information constitutes the file system **"Meta-data"** [24]. As a general definition, Meta-Data is usually anything which characterize one (or more) "*file*". For a file system, there are several pieces of information about a file which are meta-data: for example, the owner, security access controls, date of last modification, creation time, and size which belongs usually to low-level file system operations. Since meta-data is not uniquely defined, in the scientific physics world, as meta-data we also count information such as: production series, the collision, the beam energy and any external conditions not saved in the stream of byte but leading to the existence of the files. Sometimes, information such as number of events or the "triggers", accessible by inspecting the content of a file, are also defined as meta-data for easier definition of collection of files (or data-sets).
One can imagine, that a file can have multiple copies and we refer them as **"replicas"**. Therefore, in data management, one defines a unique result of meta-data query as **Logical file name (LFN)**, and speaks of a "*LFN space*", while the physical nature or occurrence of a file are **Physical file name (PFN)** defining a "*PFN space*" with many replicas of a file. The relationship between LFN and PFN is 1:N,

where N is greater or equal one. In order to distinguish the relations between all defined terms, a concept of Meta-Data Catalog, File Catalog and Replica Catalog were introduced corresponding to Meta-Data look-ups, leading to sets of logical file names each of which are associated to one or more physical file names. Each can have separated implementation to achieve different requirements and needs.

To this aim, STAR had developed a scalable and reliable File catalog [25], [26] that not only holds information about physical location of a file but also meta-data information of the file grouped into logical file such as numbers of events, triggers etc. being used by its users on a daily basis to identify accurate data-sets which is just a collection of files. The structure of the Catalog respects the basic layout of the three layer separation (Meta-Data, PFN and LFN) but presents itself to the user as a single and flexible API and command line tool allowing any queries in any name space (the relations between the layers is analyzed and determined by the API).

2.5.1 Distributed data model in STAR: rootd current approach

In the Figure 2.3, we illustrate the distributed data model in the STAR environment. The environment is composed of a large set of nodes (320) with each node having from one to 3 local drives. Since the data always has a primary copy deposited by

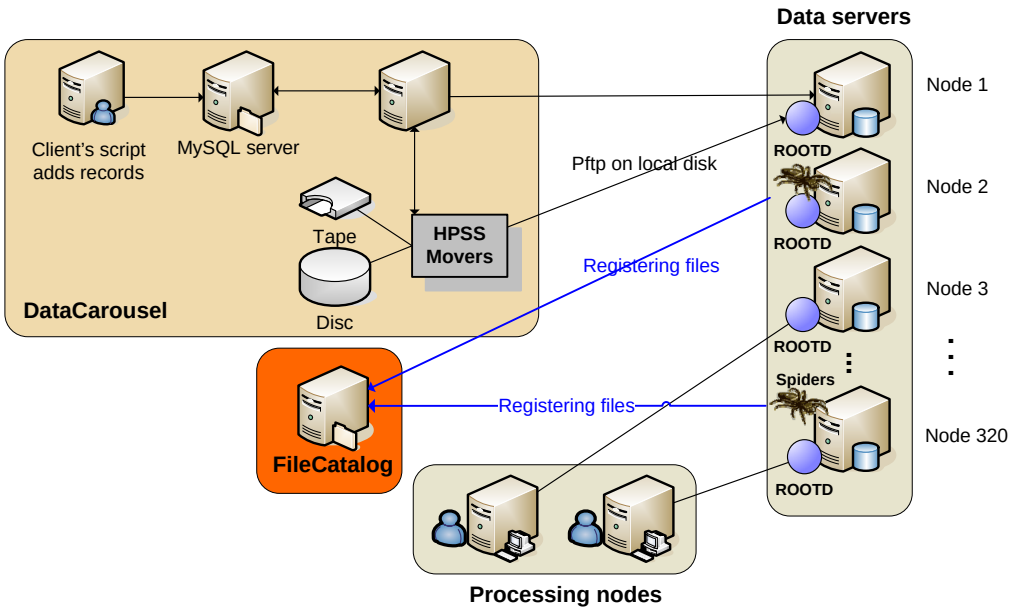


Figure 2.3: ROOTD distributed data model used in STAR experiment

the data-reconstruction process into HPSS, additional tools are needed to retrieve and populate the distributed disks in a pre-staged and static manner. To deal with this effort, the DataCarousel [25] system was developed. Its main purpose is to organize the requests made by users as well as by data population requests made on behalf of the home-grown data management tool. Such cohesive data access and

request throttling is handled by a set of compiled code interfacing with HPSS API and scripts implementing policies managing the entire system of requests therefore, preventing chaos.
In this system, requests are stored in a MySQL database and a decision making, taking into account both usage policies and file relative location on one particular tape are taken into account to therefore restrict excessive and thus much slower restore due to mounting of robotic tapes. All requests to retrieve a file are handled asynchronously.
However, the choice of where to restore the data from all available nodes is not a decision made by the DataCarousel itself its purpose only being to coordinate restore requests from HPSS. To achieve this, a set of scripts relies on space monitoring information to determine the free storage and chose amongst the many node solely based on space availability criterion coupled with the minimal set of files per storage one would need for a viable operation. Additionally, Spiders [25] keeps track of newly appearing files on each storage element and add them to the replica catalog. The Spiders also detects data which disappear, the total time to update the entire catalog being of the order of minutes regardless of the number of nodes. Eventually, and to complete the illustrative model of Fig. 2.3, all user data-intensive batch jobs read a file remotely via ROOTD, their jobs themselves are submitted according to the selection of data-sets. The STAR Unified Meta-Scheduler (SUMS) [27] would resolve user's meta-data-sets into logical files and identify particular physical locations of a file using the STAR FileCatalog API. The scheduler will then split and submit a task into several jobs depending on user's job description passed during the initial user's scheduler request. This abstraction layer makes the model viable as all files in this model would otherwise be strongly associated to server and storage that is, requires exact physical location knowledge. A user would hardly be able to keep track of the data-sets and their dynamic. The system has been extremely scalable when it comes to increasing the number of data servers, including its back-end catalog and its relative accuracy over a ten million replicas size problem.

But while sophistication and faultless features could be achieved at a first glance, the system still has its major flaws and deficiencies. The biggest is the lack of dynamic features as files are added and removed. ROOTD being by essence Physical File Name (PFN) oriented, it first needs constant cataloging and therefore the system lacks the flexibility of moving the data around without special handling. Even though the files would be distributed at multiple places, physical file access requires exact reference at submission: by the time the job really starts, the entire load picture of the cluster may very well be different from what was used for the file access decision making process. Files placed on overloaded and not responding nodes could suddenly be requested and the scheduled job would die. This is inherent to the latency between a job dispatching and the time the work unit to really starts, this cannot be circumvented within a PFN model. In fact, another of those problems comes when a node suddenly re-appears but the disk holding the data was wiped-clean (maintenance downtime due to disk failure and replacement). In such cases, the *Spider* do not only have little time to update its information but may not even exists since the system disk was wiped out (and so would be the automatic start-up scripts). Once again, the jobs were already scheduled with the

previous knowledge of file present on that storage; this would be fatal to a job. More obvious, the data population is relatively static: users could access only the data-sets already pre-populated in the system but never have a chance to access data-sets available on the mass storage. A dynamic system must therefore have the capability to hand shake with mass storage systems. Finally, a more subtle consideration, no authorization mechanism exists in this system as there lacks write access and advanced authorization layers. Also, such system should be self-adaptive, relying on its own coordination mechanism to balance load and access rather than relying on an external component providing mapping from meta-data or logical to physical name space.

2.6 Distributed file systems for HENP environment

In contrast to central server designs and recent discussion of hardware and software postulates, one can imagine building a truly distributed network file system with no central bottleneck or single point of failure. The purpose of a distributed file system is to allow users of physically distributed computers to share data and storage resources by using common files system. The main and basic features such a system must accomplish are [28], [29]:

- **Data consistency**: distributed file systems operate by allowing a user on a computer connected to a network to access and modify data stored in files on another computer. Thus a mechanism must be provided in order to ensure that each user can see changes that others are making to their copies of data
- **Uniform access**: a distributed computing environment should support global file names. One mechanism that allows the name of a file to look the same on all computers is called a uniform name space
- **Security**: distributed file systems must provide authentication. Furthermore, once users are authenticated, the system must ensure that the performed operations are permitted on the resources accessed. This process is called authorization
- **Reliability**: the distributed file system scheme itself improves the reliability because it's distributed nature, that is, the elimination of the single point of failure of non-distributed systems
- **Availability**: a distributed file system must allow systems administrators to perform routine maintenance while the file server is in operation, without disrupting the user's routines
- **Performance**: the network is considerably slower than the internal buses. Therefore, the fewer clients have to access servers, the more performance can be achieved by each one

- **Scalability**: the performance of the distributed file system must scale with number of clients and servers
- **Standard conformance**: comply with the IEEE POSIX 1003.1 file system application interface (C API)

The requirements coming from High Energy and Nuclear Physics (HENP) environment, gathered up by many years of experience and followed by recent computing directions and infrastructure such as Grid [30], are [31], [32]:

- **Fault tolerance**: a high degree of fault tolerance at the user side is mandatory to minimize the number of jobs/applications failure after a transient or partial server side problem or any kind of network glitch or damaged files
- **Load balancing**: a load balancing mechanism is needed, in order to efficiently distribute the load between clusters of servers and preventing hot spots in cluster
- **Tertiary Storage integration**: in order to support incredible amount of data, mass storage system integration is required
- **Grid support**: a distributed file system should have the ability to connect to other instances located in different parts of the world. It should have a capability to share and interchange data with other storage solutions

There are currently 2 distributed file systems partially complying with these requirements, well known in HENP computing: dCache [33], [31], [34], [35] and Xrootd [32], [36], [37].

2.6.1 The dCache system

The **dCache system** is a sophisticated system which allows transparent access to files on disk or stored on magnetic tape drives in tertiary storage systems. It is jointly developed by DESY [38] and Fermilab [39].

dCache has proved to be capable of managing the storage and exchange of terabytes of data, transparently distributed among dozens of disk storage nodes. One of the key design features is that, although the location and multiplicity of data is autonomously determined by the system, based on configuration, CPU load and disk space, the name space is uniquely represented in a single file system tree. The system has shown to significantly improve the efficiency of connected tape storage systems, through caching, i.e. gather & flush, and scheduled staging techniques. Furthermore, it optimizes the throughput to and from data clients as well as smoothing the load of the connected disk storage nodes by dynamically replicating files upon the detection of hot spots. The system is tolerant against failures of its data servers, allowing administrators to go for commodity disk storage components. Access to the data is provided by various ftp dialects, as well as by a native protocol (dccp), offering regular file system operations like open/read/write/seek/stat/close. The dCache name-space (*pnfs*) from the user perspective looks like any other cross

mounted file system. Furthermore the software has an implementation of the Storage Resource Manager protocol, *SRM* - SRM is an open standard for grid middle-ware to communicate with site specific storage fabrics.

The dCache name-space, called *PNFS* (Perfectly Normal File System) is a virtual file system that implements and simulates the tertiary storage name-space. It provides two services for dCache. Firstly it serves as mountable file system presenting the file repository. Secondly it's used by dCache as meta-data database for the file entries. This is done, by keeping the complete information in a relation database.
This implies that user doesn't need to know where a specific file is located physically. The system is maintained centrally and thus eliminated the work to be done by local system administrators while at the same time can be tuned to the need of the experiments or user groups. Since dCache is a distributed system which serves a number of disks, users, tuning will significant improve the performance of the system.

Requests to dCache may come from command-line tools like dccp or from client integrated into ROOT [22], [40]. In both cases the dCache manager is contacted through an interface called the dCache "*door*". The dCache manager determines the best source or destination *pool* or tertiary storage for the request and contacts selected pool. Finally client reconnects to the selected pool.

The pool is responsible for a contiguous disk area:

- It monitors disk space
- It holds a list of files
- It initiates the file copy process to and from tertiary storage
- It connects to data clients for the data transfer
- It monitors the total bandwidth to and from the disk area and adjusts the maximum number of movers

Clients send requests for a data file to mentioned "*door*" of dCache system. A door is a network server which performs user's authentication and forwards client requests to the pool managers. There can be more than one type of door to a dCache system, each potentially handling a distinct authentication mechanism and each perhaps residing on a separate host. The concept of Doors allows having multiple instances of one same kind of door running on different hosts for load sharing and failing safeness.
The Figure 2.4 shows dCache architecture and handling of a request. In this figure, we have represented the architecture and highlighted 2 main single points of failure as well as one performance bottleneck. Each request needs to communicate with the admin node which contacts the pnfs manager to obtain meta-data information of a file and also the location of a file within the pool. Holding one admin node with each dCache sub-systems is very dangerous and represents single point of failure. The performance bottleneck resides in the fact that pnfs database is implemented as a relational database with a limitation of scalability and performance. This performance hit has been observed in dCache deployments composed of large number of clients.

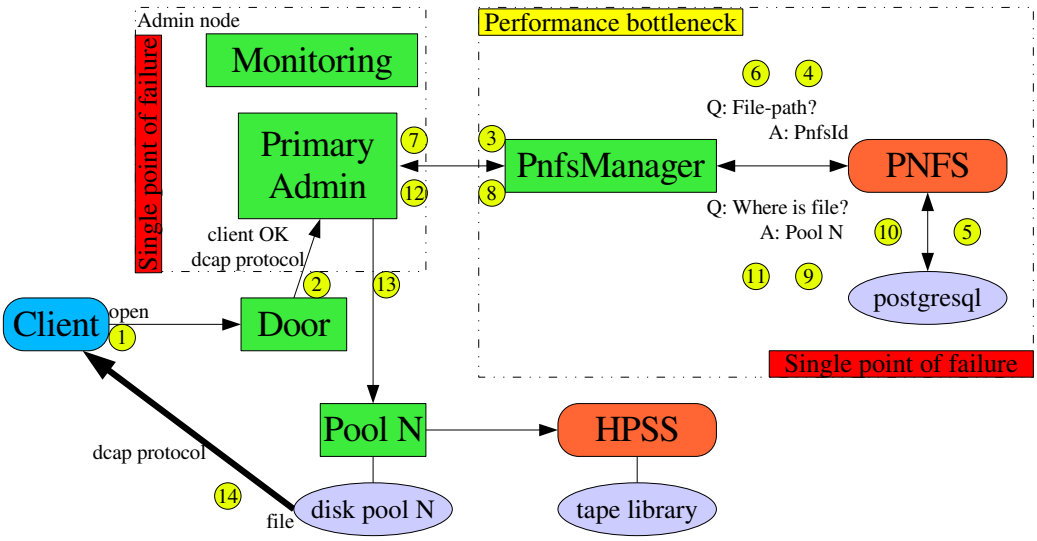


Figure 2.4: DCache architecture overview with single point of failures

2.6.2 Xrootd (eXtended rootd) system

The previous solution has proved that relying on complete knowledge of where every file resides in the system (i.e shape of comprehensive internal catalog component) is not generally a scalable and a high performance solution. The management overhead of meta-data and location of data is significant and dependent in some way on the size of the repository.
Moreover, a distributed data access architecture based on the concept of centralized catalog (*PNFS in dCache*) is more exposed to the risk of having an additional single point of failure for the whole system. This leads to another type of solution: adopting a "*publish model*".
It inheres on the view that users can write using appropriate system and then publishes the file for read access. Once file is published it can be only deleted, never replaced with identically named file with different content. This model corresponds quite well to the environment of handling scientific data: write once, publish, and read many times. One can easily imagine that the publishing of a file can be achieved with external processes such as the STAR File Catalog described in section 2.5. Externally, the performance of the catalog could be easily achieved by making the database distributed among multiple servers and load balanced. It is however disconnected from xrootd so both components could be self reliable and eventually improved.

All requirements listed at the beginning of this section and additional requirement of external cataloging complies to the *eXtended rootd system* also known as **xrootd** [36], [32], [37], [41], [42]. Its architecture allows the construction of single server data access sites up to load balanced environments and structured peer-to-peer deployments, in which many servers cooperate to give an exported uniform name-space.

If we compare side to side rootd and how xrootd system can solve our problems with its architecture and features, we arrive at the following conclusions. First,

ROOTD knows only about PFN forcing a linear scaling of the catalog as the number of data servers increase: XROOTD on the contrary knows about LFN, and refer to LFN for any data located within the xrootd system; there is no need for external additional cataloging procedure of file locations or Spiders. XROOTD load balancing mechanism determines which server is the best for client's request to open a file; nodes are selected based on reported information such as load, network I/O, memory usage and available space ensuring that the scenario of an non-responsive node would never occur. XROOTD additionally has fault tolerance features and load could be taken by other data-servers holding the data shall one data server be offline. But additionally, XROOTD implements a plug-in to interact with mass storage. Missing data can be again restored from MSS within the user's job and a delayed job introduced by dispatching with incompatible portrait of available files is no longer an issue. If the file is not present by the time the job starts, it will be imported into the xrootd system space again. Within the same feature, one could imagine a completely dynamic data space population, no longer relying on pre-staged data but on a mechanism of "data on demand". As users request new data, it appears in the system unlike our rootd based system where data-sets have to be judiciously chosen before hand. Finally, XROOTD has a plethora of authorization plug-in which resolves the "trusted/untrusted" write access issue hinted earlier and open avenues to a finer access mechanism granularity.

Table 2.1: Distributed systems comparison

	ROOTD	**DCACHE**	**XROOTD**
Developed by	ROOT	DESY & FNAL	SLAC, BNL
Scalability	no limits	small farms	no limits
Security	any auth	any auth	any auth
Platforms	all platforms	all platforms	all platforms
Fault-tolerance	No	MSS plugin	MSS plugin
MSS plugin	No	Yes	Yes
Authorization	No	Yes	Yes
Load balancing	No	No	Yes
Protocol	No	dCap	xroot
Grid integration	No	SRM (frontend)	SRM (frontend)
Single point of failure	Potentially each node(but none in particular as disconnected)	Name-space handling, head node	No

As a summary, table 2.1 provides partial overview on comparison of searched and mentioned requirements.

We will now give a quick overview of the xrootd.
One of the basic component of the system is a daemon called **xrootd**. The main purposes of this component are:

- An implementation of the functionalities of a generic file server (such as open, read, seek etc.) and therefore providing byte level access to any type of file

- An implementation of the extensive fault recovery protocol that allows data transfers to be restarted at an alternate server
- An implementation of a full authentication framework
- An implementation of an element that allow xrootd servers to be clustered together while still providing a uniform name-space
- A communication optimizations such as handling of asynchronous requests, network scheduling and thread management

While the xrootd server was written to provide single point data access performance with an eye to robustness; it is not sufficient for large scale installations. Single point data access inevitably suffers from overloads and failures due to conditions outside the control of one server.
The approach to solving this problem involves aggregating multiple xrootd servers to provide a single storage image with the ability to dynamically reconfigure client connections to route data requests around server failures. Such an approach can work as long as servers are not independent. That is, servers can be aggregated and a failure of any server can not affect the functioning of other servers that participate in the scheme. This model is close in philosophy to the one of the peer-to-peer [43] systems which have shown to be extremely tolerant of failures and scale well to thousands of participating nodes. The second component of the Xrootd system is a daemon

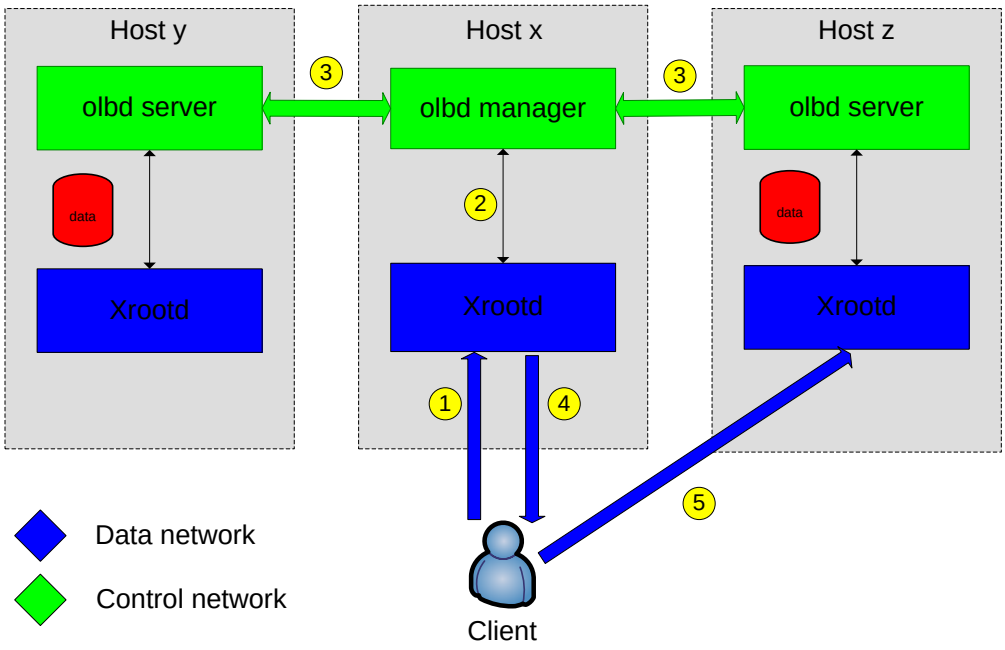


Figure 2.5: Schematic interaction of xrootd and olbd

called *olbd* (Open Load Balancer Daemon). It is a specialized server that can provide information to the Xrootd systems and steer clients toward appropriate servers (least loaded). In essence, the whole system consists of:

- A logical data network (the xrootd servers)

- A logical control network (the olbd servers)

The control network, as shown in Figure 2.5, is used to cluster servers while the data network is used to deliver actual data to the clients. The definition of a node in the Xrootd system is a server pairing an xrootd with an olbd. An olbd can assume multiple roles, depending on the nature of the task. In a *manager role*, the olbd discovers the best server for a client file request and co-ordinates the organization of a cluster. In a server role, the olbd provides sufficient information to its manager olbd so that it can properly select a data server for a client request. A *server role* of olbd is essentially a static agent running on a data server node. In a *supervisor role*, the olbd assumes the duties of both manager and server. As a manager, it allows server olbds to cluster around it, aggregates the information provided by the server olbds and forwards the information to its manager olbd.

As shown in Figure 2.6, the system is organized into a B-64 tree structure,

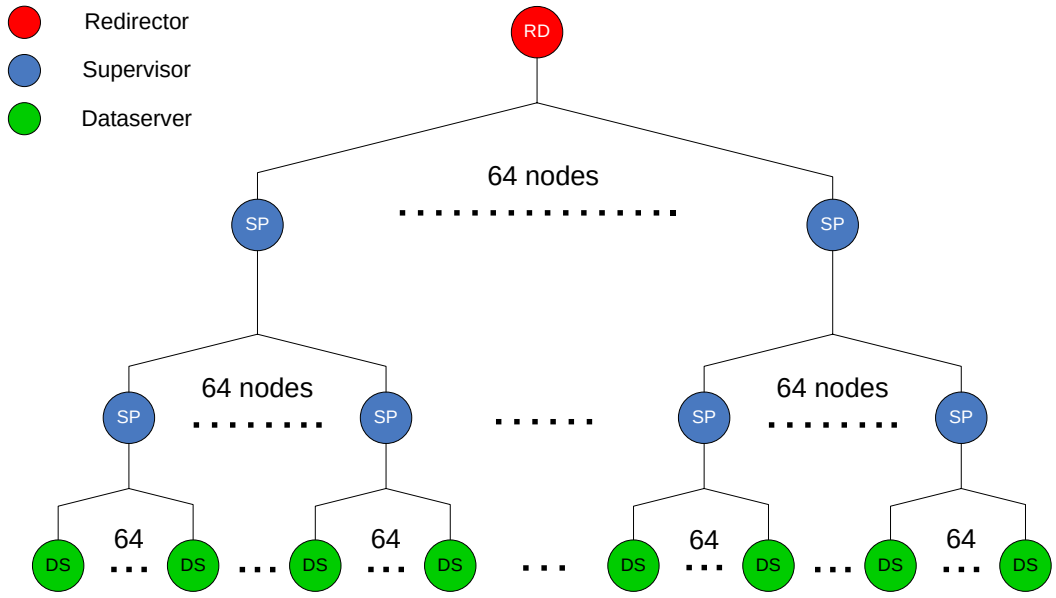


Figure 2.6: Example of B-64 tree structure used for clustering xrootd servers

with a manager olbd sitting at the root of the tree, sometime called redirector. The redirector tasks is essentially to redirect client to xrootd data-servers. A hierarchical organization of 64 size cells provides the ability of using fast 64-bit operations for selecting a server in a sub-tree. Additionally, it scales well with minimal message overhead being broad-casted to all servers.
By simple formula, two-level three (1 level of supervisors and 1 level of data-servers) can route up to 64^2=4096 servers. One can start to argue, that this number could be reached at some point. To pretend any server limitations, xrootd offers ability to have multiple supervisor levels in the tree. For example three-level tree seen at the figure (2 levels of supervisors + 1 level of data-servers) can route up to 262,144 servers. This number can be consider as human management limit of such amount of hardware at one place. However, theoretically the architecture design can serve infinite number of servers.

Initially, each node contacts the manager and requests a service slot. If the manager is full (i.e., already has 64 nodes reporting to it), it redirects the incoming node to the supervisor nodes that are currently subscribed. The redirected node then attempts to find a free slot at one of the supervisor nodes. Full supervisors will, in turn, redirect the incoming node to their supervisor son nodes. Supervisor nodes have priority over server nodes and displace any server node occupying a slot in a fully subscribed node. This algorithm builds a tree that spans all the nodes and practically configures the nodes into a B-64 tree of minimal height with supervisor nodes placed as close as possible to the manager node.
One can simply take a hint that the root node of the tree is a single point of failure in the whole architecture. In order to provide full redundancy, multiple redirectors could be set up with fail-over mechanism. Moreover, one can easily set up a DNS round robin mechanism over multiple redirector nodes to provide uniform access for clients and also providing load balancing at the root level of the tree being several times cloned. In this case, each supervisor has to connect to each manager at the root head to create connection mesh environment as shown on the figure 3.7, where DNS round robin serves as a simple load balancer between redirectors.
This makes XROOTD an excellent high-performance, extremely fault-tolerant and scalable solution for serving STAR physics data with no single point of failure.

Chapter 3

Delivering high-performance and scalable data access

Our primary goal and core of our work was the deployment and evaluation of the Xrootd system for STAR physics data at the RHIC Computing facility. The purpose of this evaluation was to answer a question of high performance and scalability of this system in very large deployment. While xrootd has been tested and adopted in many academic organizations which have the necessities of massive data access, their installations do not exceed more than 100 data servers and do not serve PetaBytes of data. Moreover, our installation is not only unique with its magnitude, but also with using shared concepts of environment not been seen in other installations. In other words, we are using xrootd on non-dedicated hardware for storage purposes, i.e. machines serving data could be also used for computational needs (usually user's jobs continuously analyzing data).

In the next sections, we will present all details about initial deployment offering basic access to data, through enabling access to the Mass storage system (HPSS at BNL), tuning I/O from MSS for the best performance and finishing at monitoring the behavior of the whole system. The last and final section of the chapter describes the measurements of aggregate I/O comparing to several storage solutions for having a baseline evaluation of whole system in real-life environment.

3.1 Basic xrootd deployment

As we mentioned in previous chapter, each host needs to serve two daemons: xrootd for data access and olbd for the purpose of load balancing and aggregating all servers together. The xrootd server has its own internal structure shown in Fig. 3.1. It is composed of multiple components, each component serves a discreet task:

- **xrd** - provides networking support, thread management and protocol scheduling
- **xroot** - implements the xrootd protocol
- **ofs** (open file system) - serves as multi-component coordinator (odc, oss)
- **odc** (open distributed cache) - has main task to communicate with olbd through the file socket
- **oss** (open storage system) - provides access to the underlying file system (actual I/O, meta-data operation)
- **acc**(access control) - consists of the two separated sub-components:
  - ◇ *Authentication* - provides the verification of the identity of a person
  - ◇ *Authorization* - grants an access control privileges to the user

Each of these components could be separably configured using its configuration reference [44], [45], [46].
These references contain several tuning possibilities and features like type of

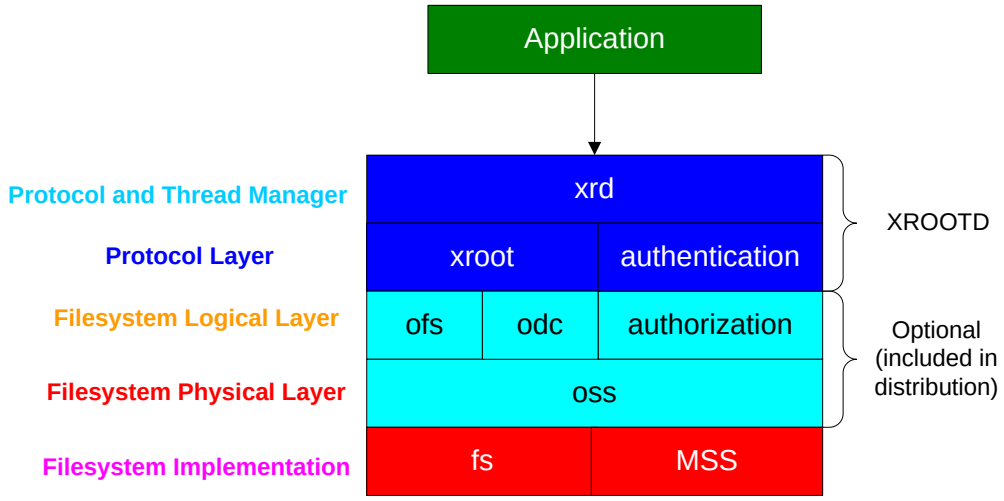


Figure 3.1: Xrootd architecture components

authentication, number of threads devoted for request serving, port numbers for listening to requests etc. Initially, one has to choose his basic configuration directives such as port numbers, paths to be exposed to a user etc. and put them into one configuration file which will define his system. This file is passed to the xrootd binary during execution. Olbd server has the same logic and is easily configurable via its configuration directive [47]. Since each xrootd component has its own defined prefix (acc, xrd ...), all configuration directives can be hold and maintained in one configuration file.
For dealing with different xrootd roles (such as redirector, supervisor and data-server), xrootd offers a feature called "*instance names*". The instance name is passed to the binary executable and could further be used to select, within a single configuration file with if statements like syntax capabilities different configuration directive for different roles. The implementation of minimal logical language within a configuration file is a very powerful concept making the maintenance of a single configuration file easier regardless of the roles.

Example:

- executable:
  xrootd -c /home/pjakl/bla.conf -n redirector -l /home/pjakl/bla.log
- configuration file:
  if named redirector
  xrd.port 1095
  fi

Figure 3.2 shows our first basic installation with one redirector node for 320 data-servers and 6 supervisor's nodes. The number of supervisors is counted as a fraction of the number of all data-servers and the size of the cell in a B64 tree. In our case, we have 320 data-servers; 64 data-servers per supervisor would lead to the need of at least 5 supervisors, the sixth one, taking the role of the fail-safe margin, i.e being

able to smoothly transfer a workload of data-servers subscribed to a newly "*crashed*" supervisor node if ever happen. There is no real limit on the number of supervisors so, in other environment, the margin could be adjusted depending on stability of the system as a whole. The first and basic functionality of the Xrootd system is to

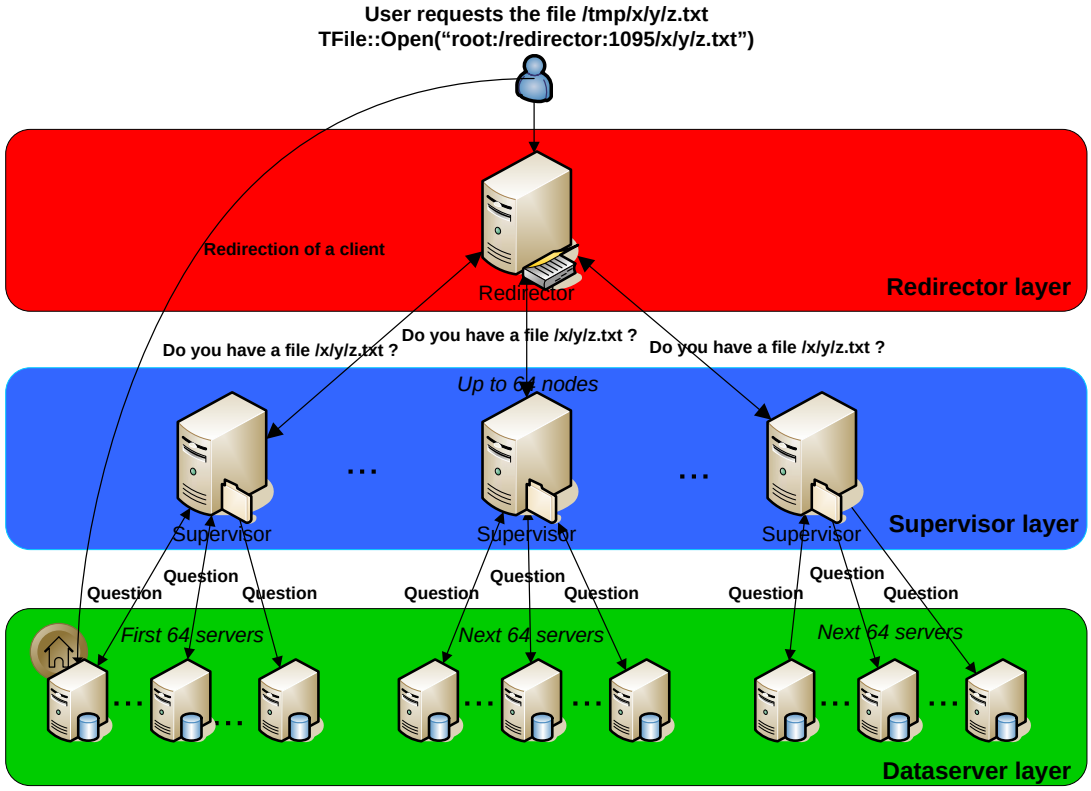


Figure 3.2: The overview of the basic xrootd installation

provide an access to files already placed on nodes. From the figure 3.2, it is clear, that client is steering its request to a redirector node. This root node broadcasts messages to other nodes with a simple question: "*Do you have the file XX??*". The supervisor's nodes serve as intermediate managers and shift the message lower to the tree. When the question is answered, the client is redirected to the particular node holding the data. This redirection prevents from overloading the head of the structure and also allows spreading I/O load across the cluster.

One problem with our large data server deployment is also the huge amount of components which may need to be controlled as one entity. In order to perform this task as a whole a whole, for example being able to shut it down on 300 servers or proceed with an upgrade or a deployment of a testbed, we have developed a set of C-shell [48] scripts for having main functionalities to:

1. Pair the xrootd and olbd daemons with support to start,stop,restart of both daemons
2. Pass configuration and log files from different paths
3. Set instance names based on the host name of a node

4. Save the log files from local directories into a remote location
5. Set preferred operation based on the existence of the particular file

In real life, each host has a Unix crontab record where cron daemon serves as a automatic trigger of the script. This does not only offer previously mentioned ability to massively restart all servers on the cluster with creating a file at the particular location, but also provides the capability to automatically restart crashed servers in each scheduled loop of the cron daemon whenever it happens.

3.2 Enabling Mass Storage System (MSS) access

From a basic installation users don't have an access to files stored on tapes of MSS or the system in not able to replicate files elsewhere in the case of overloaded and not responding node.

We can implement those features by enabling and integrating Mass storage system access inside our basic installation of xrootd. To achieve this, the Xrootd systems provides two plug-in implemented as external processes (script, code ...) requiring specific arguments and expected to return a well defined sequence of values and a completion status. The internal interfaces are:

- **mssgwcmd** - the command responsible for communication with a Mass Storage System in order to perform meta-data operations such as stat, directory list etc.
- **stagecmd** - command responsible for bringing files from the Mass Storage System into the local disk cache

We have implemented mssgwcmd command as the Perl script using FTP Perl module for communicating with HPSS FTP server. The second script was also implemented as the Perl script using the get FTP command for bringing a file from HPSS into local disk cache of the server.
The figure 3.3 shows an overview of the xrootd installation with the added MSS layer. When using MSS access, the whole work-flow of request handling is changed. As first, all servers has to answer the same question "*Do you have the file XX ?*". But if none of the data-servers have the file (or are not able to serve the file due to a high load) the file is scheduled to be staged from mass storage (HPSS in our case).
For the full cycle to complete, xrootd has to firstly choose the best node for serving a request. When the node is selected, the mssgwcmd command is executed with the option to stat a file for the verification of the file's existence inside MSS. If this is passed successfully, the stagecmd is triggered to bring the file.
The definition of what constitute a "*best*" node is arbitrary and the selection algorithm will be explained in more details at the section 3.3.

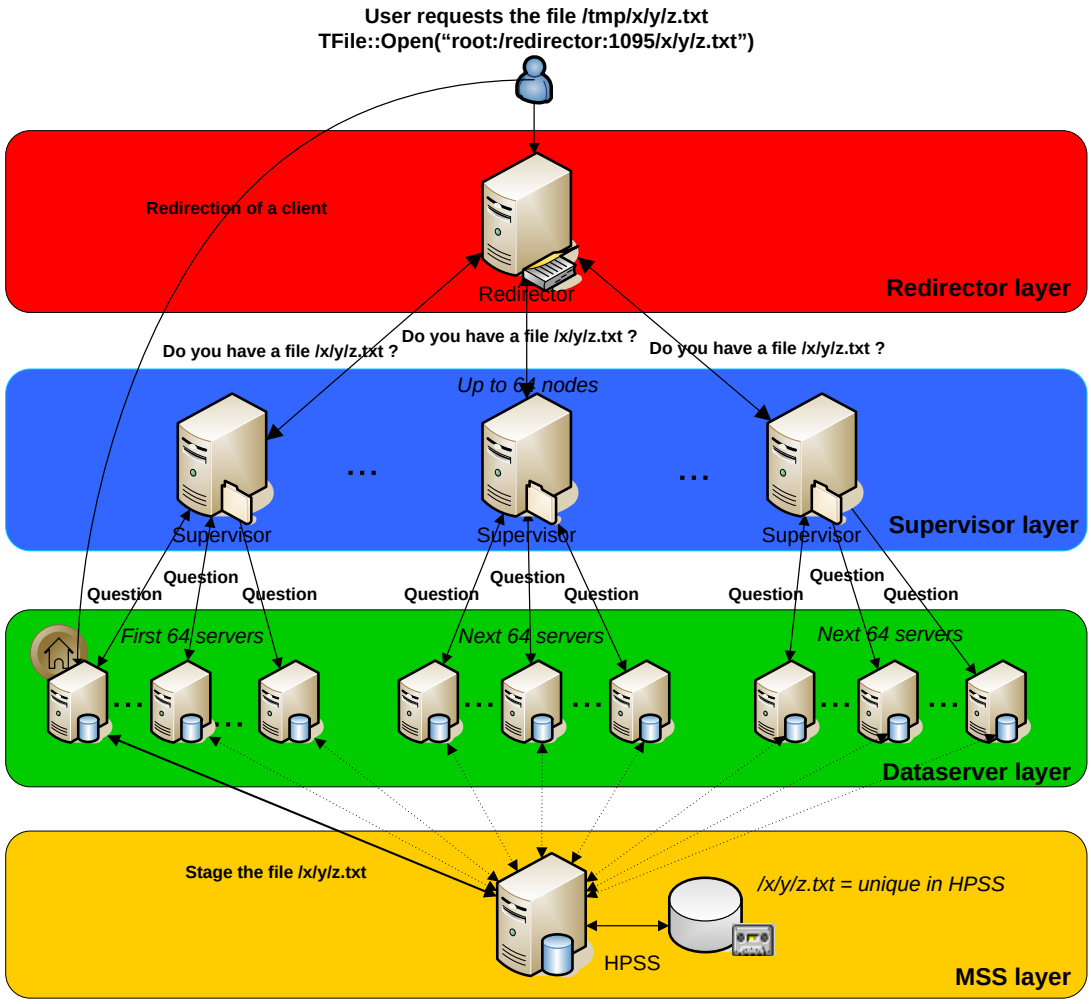


Figure 3.3: The overview of the xrootd installation with MSS access

3.2.1 Un-coordinated MSS requests

During the testing of MSS plugin, we performed a scalability test and have observed that many data-servers may request data simultaneously, therefore fall into the uncoordinated request trap. Figure 3.3 shows that each server in the cluster can connect to HPSS in an uncontrollable manner and without any sophisticated coordination between them.

By essence, HPSS is a robotic based machine with main task of mounting tapes into a finite number of available drives where data can be read and transfer on the disk cache. While the cache can be made larger enough no nor be a concern, each request can come with the request for a file on a different tape and without any other guidance, HPSS has to mount each tape separately for each file which can lead to a excessive mounting of tapes as showed on Figure 3.4. Additionally, and since all requests from our data-servers may come simultaneously, the number of the ftp connections can accumulate as a linear function of the number of requests. This is seen on Figure 3.5. Both problems lead to HPSS collapse and deadlock situation, system ends up with zero aggregate throughput IO as seen on Figure 3.6. To

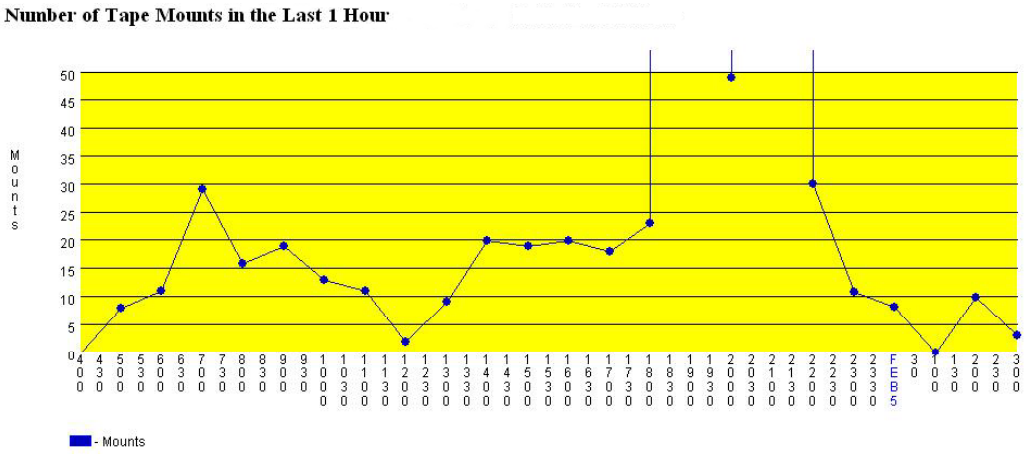


Figure 3.4: The exceeding of the tape mounting

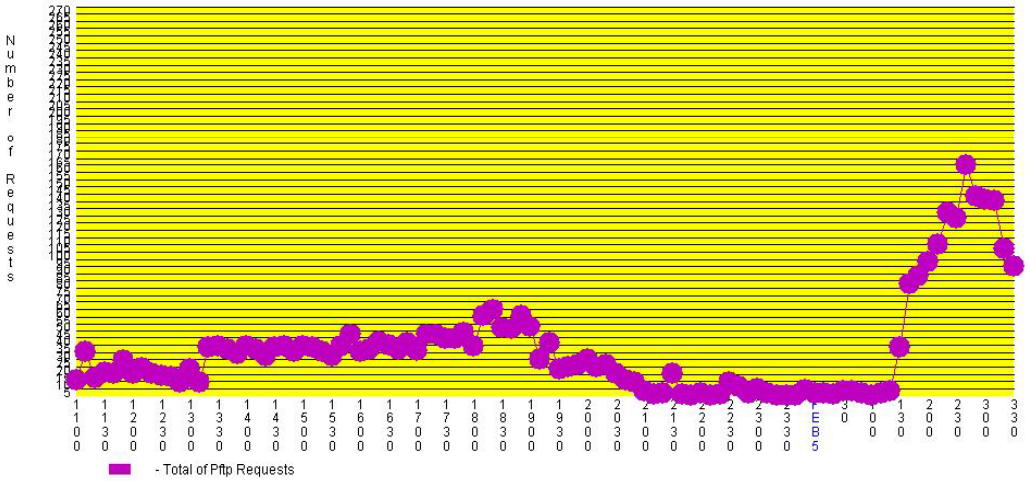


Figure 3.5: The exceeding of the ftp connections

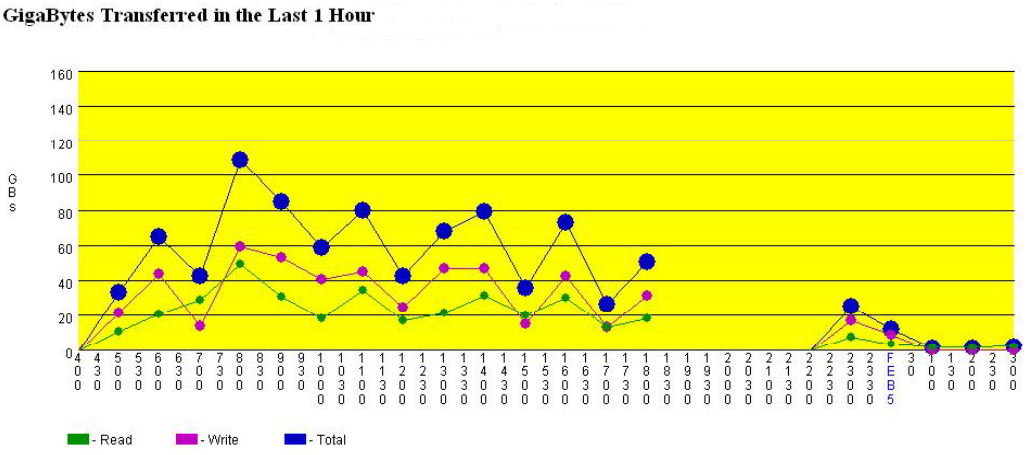


Figure 3.6: The aggregate IO from/in HPSS

overcome this problem which by itself, would not allow to enable the HPSS plugin large data server environment, one has to search for a solution with main features like:

- Coordination and queuing of requests
- Sharing access with other data management tools involving policy based authorization with different priorities per user or group
- Sorting of file requests with effort of having as many file requests as possible on the same tape and minimize tape mounts
- Keeping track of requests and re-queuing them in case of failure
- Asynchronous handling of requests to speed up the IO

All these requirements fulfill a system called the DataCarousel, which has been used for past few years within the STAR framework and already mentioned in section 2.5. A reader can have a objection that a single file restore within this scheme may be slower than a direct pftp (parallel ftp) approach, but this is not usable in the large scale illustrated above, even not mentioning the zero sharing of HPSS resource with other tools.

Hence, we have integrated the DataCarousel into our xrootd installation by re-writing mentioned the *stagecmd* script. The DataCarousel offering a Perl API (as a Perl module), the xrootd plugin was rewritten to exploit the DataCarousel API and pass the requests directly to the DataCarousel back-end system. The work-flow could be described by three phases:

1. Construct and pass DataCarousel API request
2. Wait until either a timeout occurs or the DataCarousel brings the file into the local cache
3. Return failure or success respectively

For simplicity, the script is just waiting for the appearance of the file in the local cache. Since all requests to the DataCarousel are handled asynchronously, it would be harder to track them in the system itself and figure out the logic depending on the carousel internals state of the request. Finally, the figure 3.7 shows the final grained architecture of our installation.

3.2.2 Creating a uniform name-space

In general, a name-space is an abstract container providing context for the items (names, or technical terms, or words). Within a given name-space all items must have unique names, although the same name may be used with a different meaning in a different name-space.

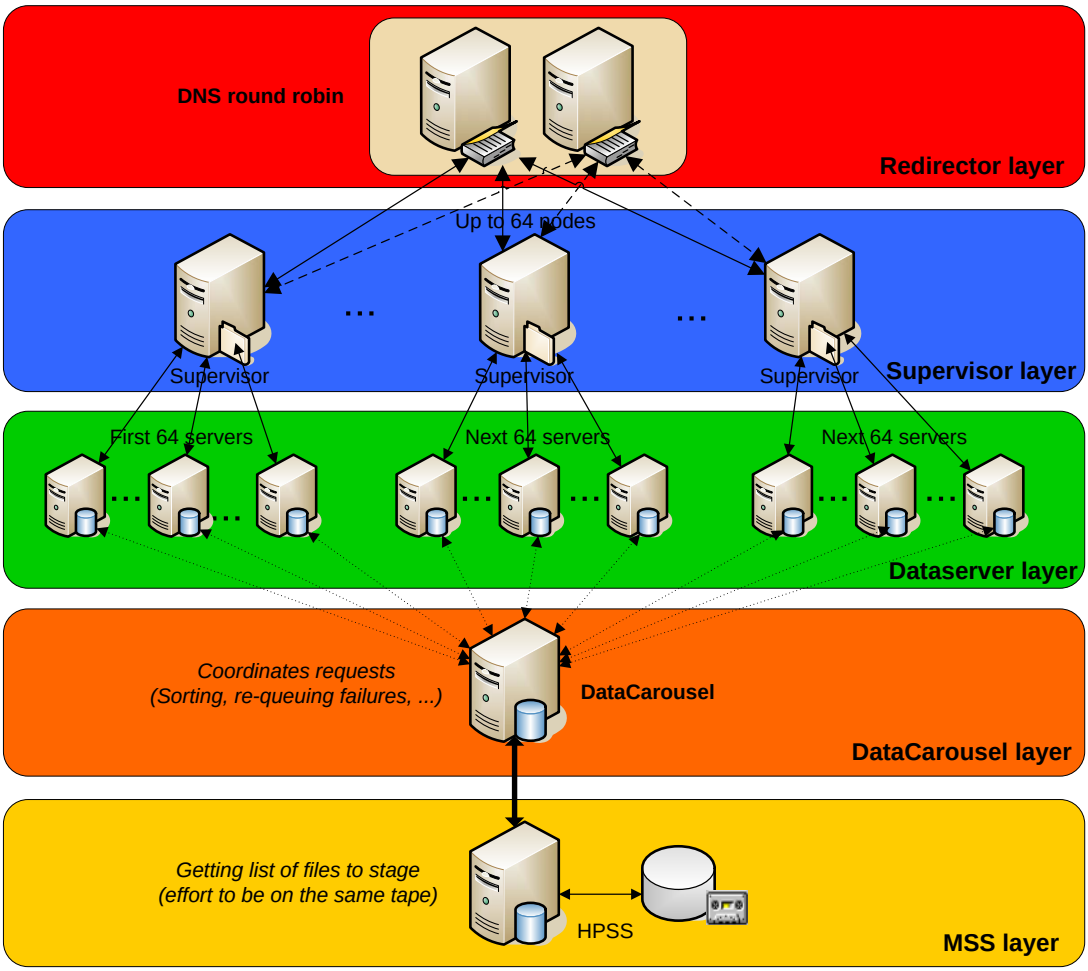


Figure 3.7: The Xrootd overview with DataCarousel integration

In the file systems, the name-space is a directory. It contains several files which must have unique name within one directory. However, one can imagine that a particular tree starting from a root prefix */xxx* could be cloned (copied) to a different device starting from a prefix */yyy*. While from a file system perspectives, the two trees together forms an aggregate name space itself a name space, files are nonetheless identical one level up down in the tree starting from either */xxx* or */yyy*. For all practical purpose, */xxx/A* and */yyy/A* may canonically be part of the "same" name-space (there is a unique transformation between the two). In general, the same name can reside in different directories located on one physical device or machine within separate base-path. In UNIX world, the mounted directories are many partitions representing physical devices (such as HDD) being attached to one machine. In global view, the fun of creating a name-space is represented by many physically independent machines within the one farm. For instance, if a farm has 1000 nodes, where each node has exactly 3 physical drives, there would be 1000 * 3 possibilities to place a file and therefore 3000 fundamentally diverse name-spaces. This could be collapsed to one unique name space by creating a transformation such as, making all separate tree appear as the same (in the same manner that stripping /xxx and /yyy would make a new A/? unique name-space).

A distributed file system has to create a single uniform name-space as, this would

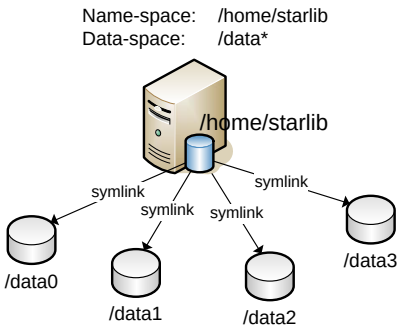


Figure 3.8: The single name-space within one node

allow removing the need to know the hardware layout detail, mount path, names to device, and therefore decrease the number of entries in a replica catalog while providing everything would be transparent to a user. But for the distributed data management system, this means that a file A would need to be checked against each element of the local device i.e. /data0/A, /data1/A etc. To overcome a case of a single name-space within one node, xrootd offers mechanism of using UNIX symbolic links managed by MPS module [49]. As shown on figure 3.8, each file has a record of a file name in name-space directory (in example /home/starlib) and a record of a file content in data-space directories (in this case $/data*$). Both of these records are together grouped using UNIX symbolic links. This approach creates an option of having large data space within single-name space at one node and reducing the number from 3000 possibilities just to 1000.

If one wants to further decrease the number of occurrence and create a single name-space spanning over all nodes of the farm, it is convenient to the concept of logical and physical files mentioned in section 2.5.

The representation of logical file names may vary depending on the representation, one possibility is to represent it as an MD5 sum of a file (name). The drawback of this solution is the fact that while LFN could be stored in the Xrootd name-space in any form it first involves an additional internal operation (a calculation, a catalog look-up for example) within each user's request.he second problem is that if the file is not found, it has to be fetched from the mass storage, which implies an additional transformation of either the primary request to the HPSS name-space or of the stored Xrootd name-space LFN into the HPSS name-space. The second alternative, is to use the LFN to match its physical location in MSS (shortly PFN(MSS)); as a single, MSS system at one site by itself ensure the uniqueness of the name of a file (unique tree). This solution offers ability to have many PFNs (xrootd server or MSS) under one reference (LFN) without any name transformation operation when request is made. In other words, shall the user access the Xrootd system using an LFN equivalent to the PFN(MSS), there would be only a trivial transformation from the user to Xrootd (one related to path manipulation as explained with /data1, /data2 and so on) and no transformation at all from the user's request to the HPSS request.

This solution also has its drawback, which assumes that there is only one MSS within one site. This becomes a problem when name-space may cover many sites spread over the world and with more than one MSS.
Therefore, we have to provide and re-architecture xrootd for having generic and flexible LFN/PFN conversion module with ability of different implementations. We designed the module as plug-able with the interface containing methods such LFN2PFN or LFN2RFN, where RFN is a location of a file in MSS. This change not only offers to use still the same logic as before, but also to create another logic applicable for name-space covering many farms on different sites with more than one MSS.

3.2.3 Coexistence with other Data management tools in STAR

In order to integrate Xrootd in the STAR environment with the least disruption, we needed to take into account the fact that ROOTD was in use within STAR framework. Since ROOTD is in essentially PFN based, we had to allow xrootd to understand both PFN-like access as well as LFN-like represented by PFN(MSS). To achieve this goal, we could simply re-use and create an implementation of the designed interface.

Figure 3.9 shows a server call stack within a user's request to open a file. User's request to open a file is in form of *XFN syntax*, we refer this as the **Xrootd File Name**, which encapsulates both scopes of different name-spaces, the ROOTD PFN-like as well as XROOTD LFN-like. Client makes a request to the server through the xrootd protocol, when request is accepted, the hunt for the file starts. The LFN/PFN module comes into the scene during the calling of the "*open_ufs*" function. The server checks firstly whether the XFN is PFN-like or secondly if it validates as a LFN. If none of these operations are successful, the file is scheduled to be staged from MSS. This involves the check whether the file is already being staged from MSS or has been seen and failed. If not, server will firstly check existence of the file in MSS using mssgwcmd command described at previous section 3.2. When the file is presented in MSS, server creates new thread devoted for staging, triggers execution of stagecmd and registers it in the list of pending requests. When the registration is done, it delays a client for the specific time. After the specified time, client repeats the initial request. When staging is successful, the server would succeed in second check of the LFN/PFN module, whether the file is presented in xrootd name-space. Our approach involved one additional UNIX stat() operation of a file and therefore didn't introduce any performance bottleneck.

3.3 Increasing I/O rate from MSS

So far, the whole work was focused on describing an effort of having a fully functional and stabilized version by starting with a basic installations, continuing with enabling

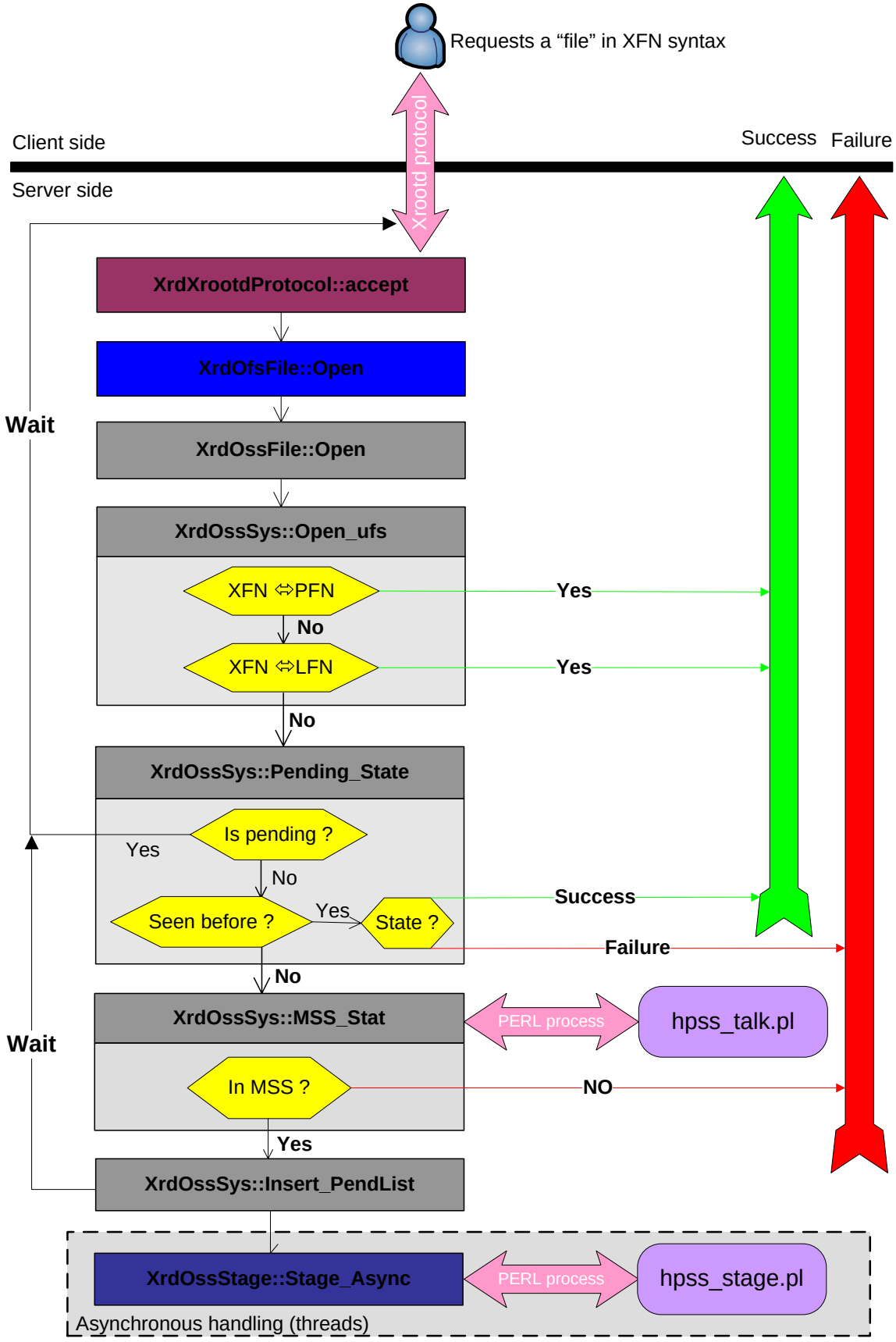


Figure 3.9: The call stack of the server side within a user's request

MSS access and its stabilization to a scalable solution in high request demand regime. The next natural step would be focus on delivering high-performance and this involves tuning of IO throughput rate within the system and from MSS.
During the testing of MSS plugin with DataCarousel, we have observed that the IO throughput from MSS is affected by selecting still the identical server several times of selection. This theory was even confirmed by trying to increase the IO throughput with increasing the number of threads devoted for staging per one node. There was no difference in IO throughput between 2 threads or 30 threads.
Thereby, we started to focus on the mechanism of selecting a server for the file restore and related topic of load balancing among the multiple choices represented by the individual servers.

3.3.1 Load balancing and server selection algorithm

Since we started to talk about selecting a node for fulfillment of requested operation, we have to directly deal with the concept of load distribution between collaborating nodes. Indeed, a distributed data access system can be in the situation where more than one choice can be available to fulfill a particular incoming request (e.g more than one replica of the file or selecting a server for the file restore).

The main purpose of load distribution is to improve the performance of the distributed system, usually in terms of response time or resources availability spread over many collaborating nodes. A side effect of this deals with the benefits coming from the distribution of the system itself, in the form of additional reliability or larger storage space or computing power. The problem of distributing a load between collaborating nodes is related to a wider concept of resource allocation. The are two main approaches for the attribution of the system load and resource allocation in distributed systems [50], **static** and **dynamic** load distribution. The static load distribution assigns a work to hosts probabilistically or deterministically, without considering the system's status or the events coming from it, where dynamic distribution monitor the workload and hosts for any factors that may affect the choice of the most appropriate assignment and distribute the work accordingly.

Static approach is useful only when the workload can be accurately characterized and where the load's scheduler is in control of all activity, or is it at least aware of a consistent background over which it makes its own distribution. If the background load is liable to fluctuations, or the characteristic of the single cooperating nodes can vary independently, some problem arise, which usually cannot be solved by means of unique static behavior.
On the other hand dynamic distribution seeks to overcome the problems of relying on how to decide which system workload may be assigned to each host for the best representation [51]. This approach tries to incorporate two factors, firstly the resources currently available at a host and secondly the resources required by the processes being distributed.

For the current available resources, this is reflected in xrootd by the definition of a "*load*" composed of 5 factors for assembled within a generic policy. Ideally,

the combination of those 5 factors will be able to reflect most common computer environments. Those are:

- **CPU usage** - percentage of cpu being used at the host
- **Memory usage** - the percentage of memory being used at the host
- **Paging usage**- the percentage of paging load being used at the host
- **Runtime usage** - the percentage of run-time usage (e.g how long the system has been running, how many users are currently logged on)
- **Network usage** - the percentage of network resource being used at the host

For resource required by the processes being distributed, it is simplest to reflect two factors related to xrootd architecture and request work-flow:

- **Number of allocations** - How many times the host was selected for a file restore
- **Number of redirection** - How many times the host was selected for opening a particular file located on a host

First consideration involves a flexible scheduling algorithm based on combination of the mentioned 5 factors, where each xrootd administrator can set up different thresholds for computing the overall workload. Indeed, these flexible computations give a power of being able to build a ideal dynamic distribution of the load among the cluster. For a realization of this approach, see the subsection 3.3.2.

There still remains the question: "*How the server is selected?*". Figure 3.10 has been designed to reflect the client interaction with the server side, mainly representing server's side work-flow of the selection algorithm.

When a client contacts a redirector node through the xrootd protocol, xrootd part of the node requests its olbd through the olbd protocol to locate a file. Olbd checks whether the file has been seen before by looking into its cache, if not and the file is new, it is added into a cache and the manager broadcasts a query for the file to all of its subscribers that have declared a capability to handle files.
The client is then asked to repeat the requests after a fixed amount of time during which responses are collected by all the managers/supervisors which propagated the message to their cell. Data-servers which has the file respond affirmatively, otherwise stay silent.
From this process, a list of 0 or more data servers holding a file is determined and when the query is resolved, two cases can happen:

1. *primary selection* - one or more replicas of the file exist on the nodes
2. *secondary selection* - no replicas on any nodes, a server has to be chosen for the file restore from MSS

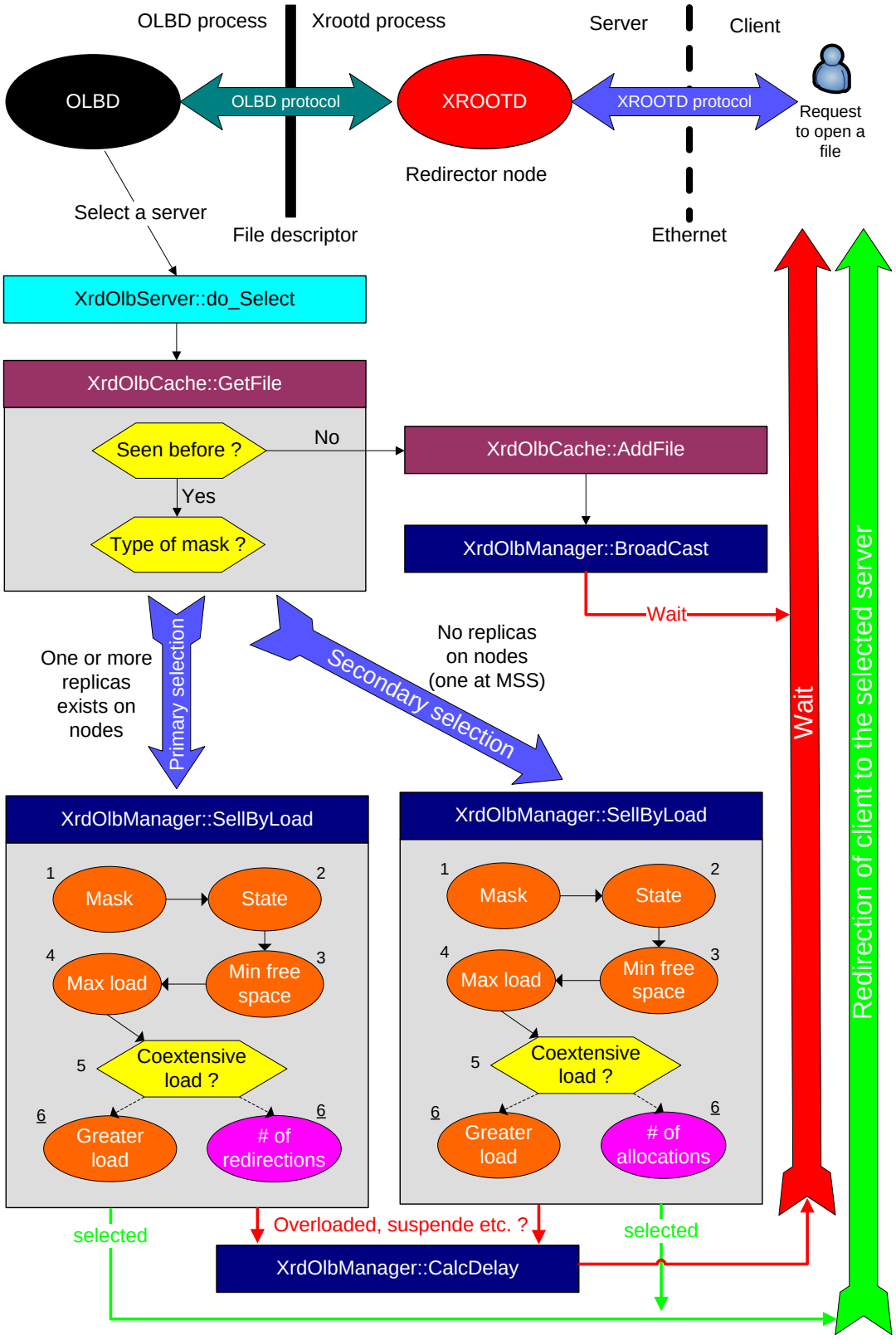


Figure 3.10: The server selection algorithm

In both cases, olbd has a result stored in server mask, which server the purpose of being able to make a selection and decision in the fastest possible way. The manager is going through all of it's subscribers in a loop and applies several conditions:

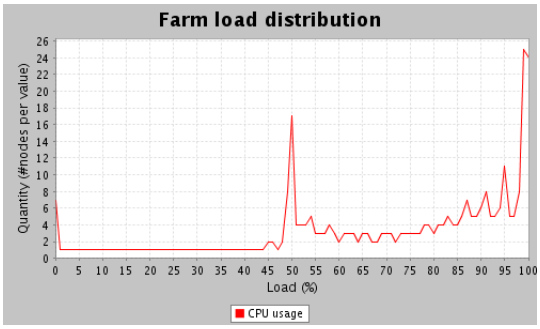
1. Mask has to match
2. Check for offline or suspended server
3. Check for minimum free space available at node
4. Check if the actual load of the server doesn't exceed the configured maximal load

If one of these conditions is false, the manager remembers it and uses this information for the final answer to the client. If the server in the loop went through all conditions without blemish and there was previously a suitable server, it would check for the coextensive load. This means whether a difference of server's loads is within a specified and configurable range. Within this configurable range (or margin), the workload of the servers are considered as identical and the manager needs to apply second factor of load distribution to select one, the second criterion is the resource required by the processes being distributed. In case of a file restore, it is either the number of allocations, or the number of re-directions. When the cycle is finished, the manager can delay a client for a fixed time computed from collected information such as number of overloaded nodes or the number of suspended nodes. Alternatively, it returns the answer with a redirection to a server.

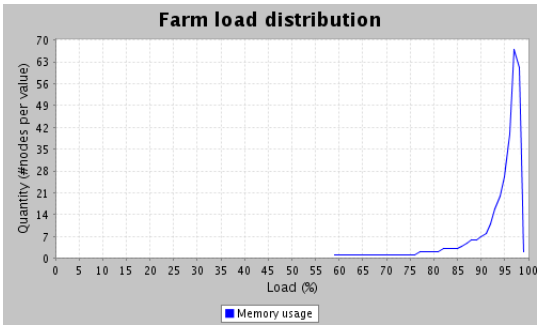
3.3.2 Investigating workload of the system

The previous section briefly discussed a general overview on how to build a flexible load distribution in the distributed file system and how this load enters into the selection of a data server. We mentioned the fact that analysis jobs are the creators of a load on the machine.
The 5 parameter space mechanism described in the previous section certainly allows to reflect most environments, e.g. CPU-bound environments may have the biggest weight on the CPU factor, memory-bound environments will have the highest benefit using the memory factor.
The fundamental questions still remains: "*How to figure out which factors-bound is my environment ?*". For resolving this issue, we have measured and collected statistic of all workload factors at each node of the Linux farm. Those results helped to assemble the final shape of the formula and determine factors-bound of the STAR environment.

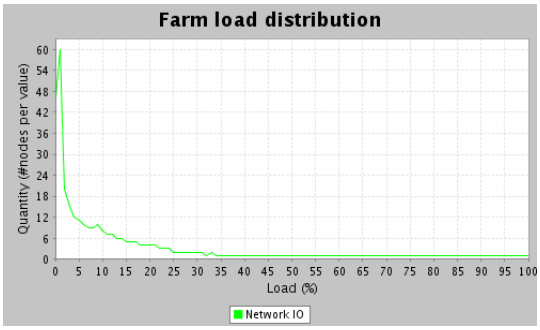
For the best representation, visualization and aggregation of the all results from the farm, we computed and prepare a plot with x-axis as the percentage of the measured load factor, while the y-axis value is a number of nodes which had the same particular value of the load. Figure 3.11 shows example plots for measurements. Basically, from those 5 factors, two of them can heavily affect IO throughput of one node, it is memory and paging factor. In most today's operating systems, the system which is lacking enough memory would start paging on the local HDD and will affect



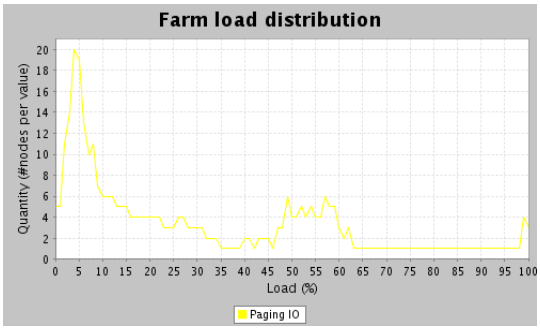
(a) CPU factor



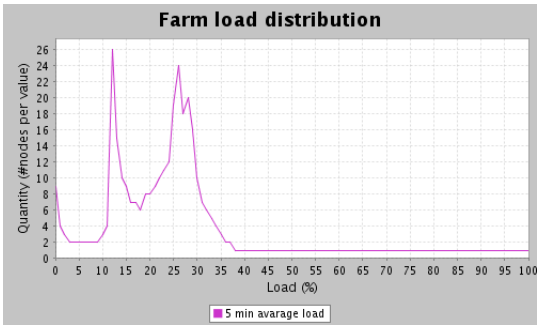
(b) Memory factor



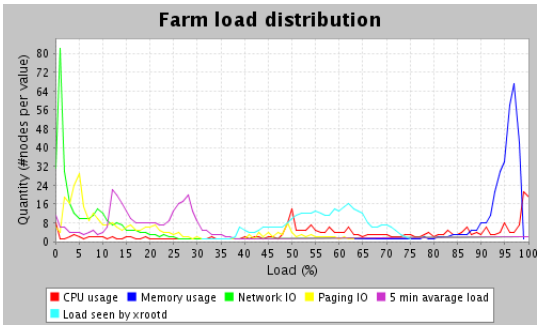
(c) Network factor



(d) Paging factor



(e) Run-time factor



(f) Overall plot with all factors

Figure 3.11: Factors-bound investigation

IO throughput being scheduled to this system. The second effect of paging could also affect the CPU itself (intense swapping could use a significant amount of resources and lead to IO thrashing). Perhaps of a lesser importance, we imagine that the factor to formula would be a CPU factor could affect the global performance as if none are available, it would impact the data server process (xrootd and olbd) themselves. Therefore as a conclusion, we assembled our empirical formula with choosing as the biggest impact on the memory, paging and cpu factors along with other consideration as follow:

- 20% of CPU factor
- 10% of network factor
- 20% of paging factor
- 10% of runtime factor
- 40% of memory factor

The network factor of the load is somehow self-correlated to the overall distribution, since using xrootd introduces higher number of network traffic by reading the file remotely through the network. We have set this threshold very low and this correlating factor will need additional work in future to determine its exact impact.

The figure 3.12 serves as the verification of the right chosen values and possible corrections and fitting. Our effort was not only assemble the load thresholds, but also to create a perfect distribution of load imminent to the Gauss distribution.
Since, the observation was provided statically and at the one moment of the farm's

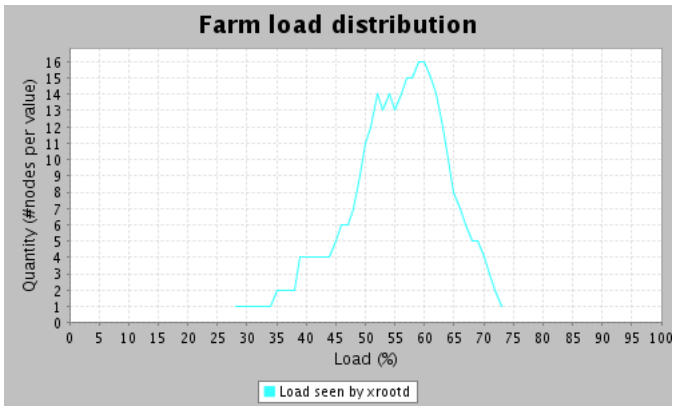


Figure 3.12: The overall workload seen by xrootd

workload, a large amount of statistical data needs to be collected to ensure a right setting of the thresholds which is stable to load fluctuations. These statistics were gathered and computed by hour, day and week periods. Additionally, to check if the overall load doesn't fluctuate too much in time, the 3-D representation of load, with the third axis being the dimension of "moving in time". It has been prepared in the same statistic's periods. The figure 3.13 shows hour statistic. For more statistic plots, please see the appendix A.

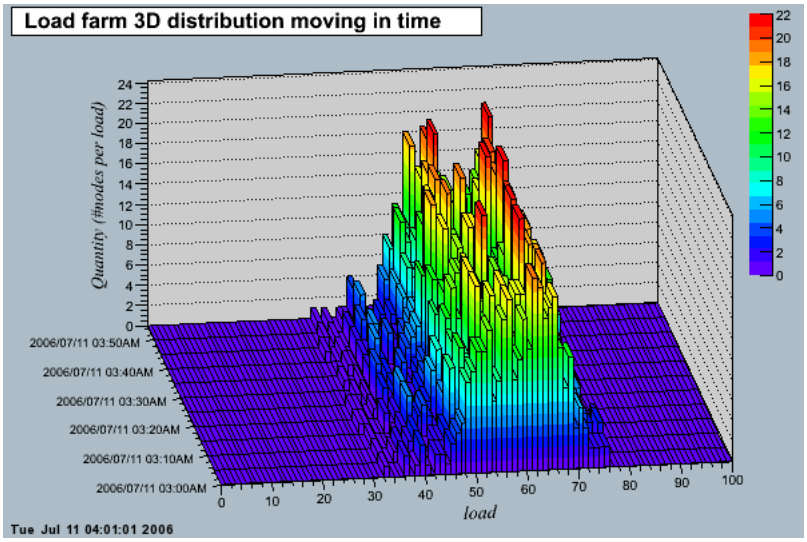


Figure 3.13: The 3-D distribution of the farm workload seen by Xrootd

3.4 Monitoring the behavior of xrootd

After the system is tuned for distributing and balancing the load with an expectation of a performance increase, one needs to be able to monitor the behavior of the system with increased number of requests.

The simplest solution is to use the system in a production mode within the STAR framework and consequently measure the scalability within user's requests. The scalability in this sense means, how the system is capable to manage a mass of requests. Moreover, preferable and interesting information would be, how the system is efficient to deal with load of requests and still being able to serve additional incoming requests without any disruption or performance degradation.

Monitoring the behavior of the distributed system involves dynamic extraction of information about the interactions among many nodes, collecting this information, and presenting it to users in useful format. Within the allocated time, we did not carry out an exhaustive study but concentrated mainly on the behavior of the MSS plugin and its efficiency.

Easily achievable information from the whole system were:

A number of all requests coming to xrootd

B number of successful requests to HPSS

C number of failed requests to HPSS

These three pieces of information helped to easily obtain and form four attractive informative plots reflecting the behavior of the system as a function of time:

1. Number of requests moving in time

2. Percentage of failures seen by the server, computed as: **C/A**
3. Percentage of failed requests over all requests to HPSS, computed as: **B/(B+C)**
4. Percentage of HPSS requests over all requests to XROOTD, computed as: **(B+C)/A**

The illustration of this effort is visible at the figure 3.14.
As an important note, users couldn't access HPSS files directly, HPSS plugin served as fault-tolerance retrieval when the node was down or didn't respond. Hence, the results shows that system is scalable, where the errors and using of HPSS plugin for file restore (file wasn't previously found on nodes) is very rare and not dependent on number of requests.

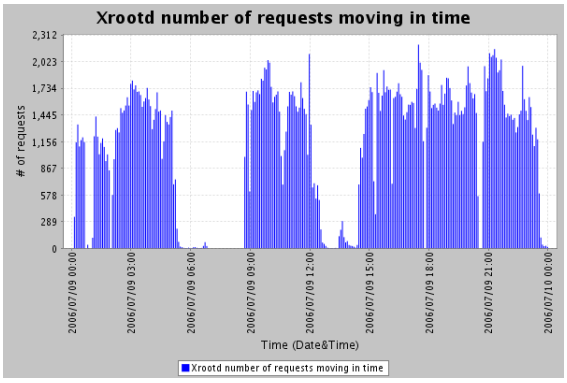
For debugging and testing purposes of the HPSS plugin, the figure 3.14(e) shows HPSS plugin errors and their relative proportions. The most significant portion of errors forms the error that file wasn't brought by the DataCarousel into disk cache within the timeout period of the xrootd plugin. Most errors occurred due to the absence of a free space which, although overall rare, are not expected events since we explained in section 3.3 that Xrootd includes a free space criteria before selecting a data-server. The understanding of this issue is scheduled for the future work.

3.5 Measuring and comparing the performance

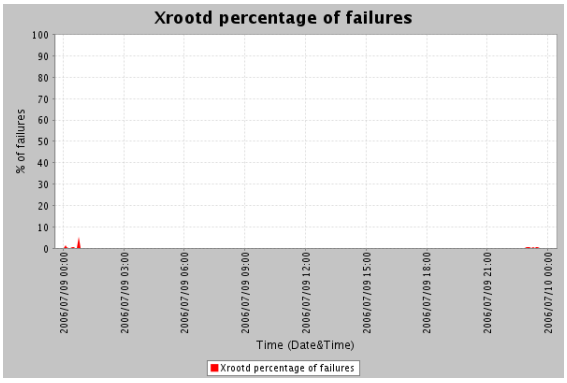
The success of the data distributed system relies on the ability to support a reasonable increasing number of users with stable performance of individual file operations and therefore achieve scalability of the system. This will certainly encourage its use and facilitate the migration of users from their addictions to other systems.

The performance of file servers or distributed systems is usually measured as a time of single operation to read/write a chunk of data, sometimes extended with many concurrent operations at one point of a time. This evidently reflects a performance of one independent server being detached from the global view of many cooperating servers in distributed environment, but not showing the aggregated performance of the whole system. It implies a need for aggregate picture of whole system in concurrent fashion and under a heavy load of many requests. Additionally, the picture needs to be taken at the same environment and under identical conditions for a comparison with other storage solutions, i.e. same kind of files (structure, compression etc.), same technique of reading, same global workload of environment etc. For achievement of all these requirements within the STAR framework, the aggregation unit was chosen as a one job reading sequentially physics events from structured files within ROOT framework. It definitely ensures requirements of the same technique of reading and same kind of files.

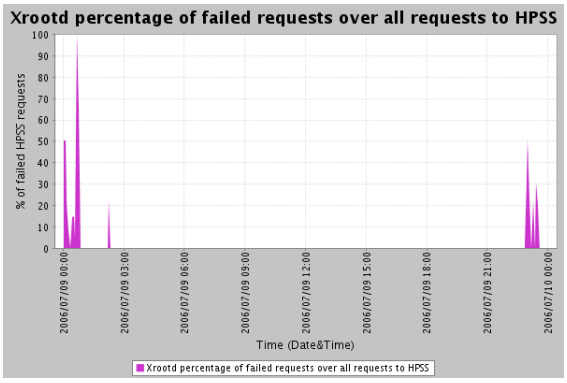
While the single server measurements are usually collected on the dedicated hardware, without any load of other processes, the effort is to see a behavior of the



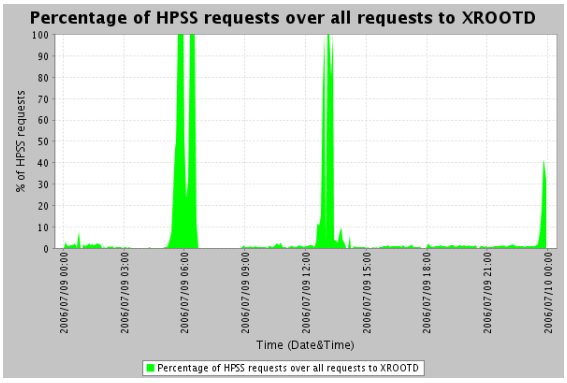
(a) Number of requests moving in time (bin=5 min)



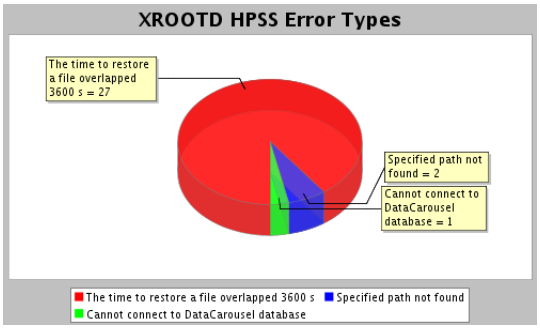
(b) Percentage of failures seen by server side of XROOTD



(c) Percentage of failed requests over all requests to HPSS



(d) Percentage of HPSS requests over all requests to XROOTD



(e) Portions and types of HPSS plugin errors

Figure 3.14: Monitoring behavior of xrootd

system in real world and scenario, i.e load caused by other users and processes. It will give a preview of the system's load-balancing efficiency. To ensure the requirement of the same global workload and load balancing efficiency, one has to measure and compute many independent tests at different times and therefore diverse workload's states of the environment.

Finally, as the global view of the performance and scalability of the system would be a fashion of increasing number of jobs and their aggregate IO throughput. To capture this picture, an additional aggregate and advanced algorithm is needed, since the view is based on many concurrent and subsequent jobs running at the same time which is very difficult to achieve in share environment. These share environments are well-known with their queues introduced in the batch systems and triggering the execution of jobs based on different conditions and policies of the particular installation.
As a summary, the measurement will produce following information:

- a starting and ending times of the mutually different jobs executed in different times of a test
- an average IO throughput of the particular job
- a job's competency to particular test of measurement
- a length of the particular test indicated by start time and end time of the first and the last collected job
- an average length of one job in the test

When each job can start and finish in different time, the aggregation of simultaneously running jobs becomes very difficult. The figure 3.15 shows 4 fundamental possibilities how the job can participate within a time range, where Θ is an average length of the job and $T_i\,, T_j$ are start, end time of a time range. To overcome the hardness of aggregating simultaneous jobs within a time range, we fetched a formula 3.1 counting a benefit of each job covering 4 mentioned possibilities within specified time range:

$$Job's\ benefit = \frac{\min\,(\,T_j\,,\ e_{job}\,) - \max\,(\,T_i\,,\ s_{job}\,)}{T_j - T_i} \tag{3.1}$$

where

(i) $T_i\,, T_j$ are start and end time of a time range

(ii) $s_{job}\,, e_{job}$ are start and end time of a job

As a proof, lets establish a case 4 as following, $s_{job} = T_i$ and $e_{job} = T_j$, which is giving:

$$Job's\ benefit = \tfrac{T_j - T_i}{T_j - T_i} = 1 * 100 = 100\%$$

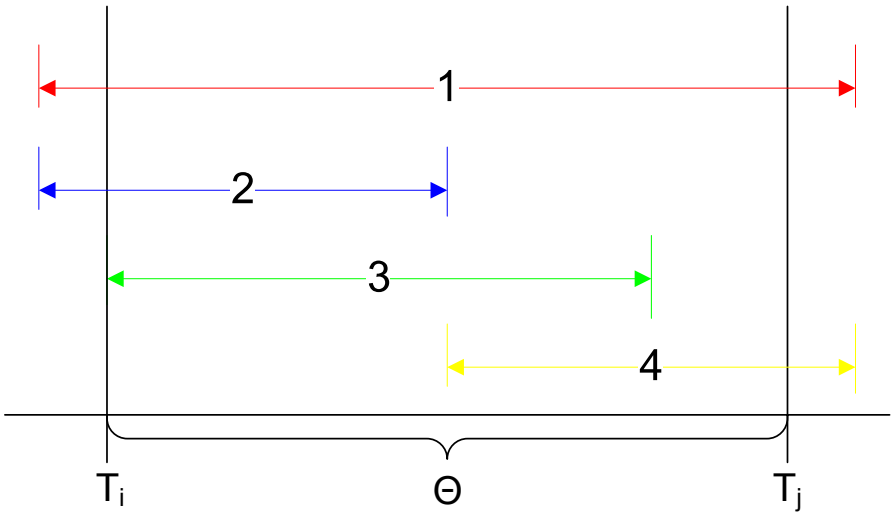


Figure 3.15: Four fundamental possibilities of job's participation within a time range

All jobs running within a specified time range could be selected by the following query:

$$(s_{job} \geq T_i \bigcap s_{job} < T_j) \bigcup (e_{job} > T_i \bigcap e_{job} \leq T_j) \bigcup$$

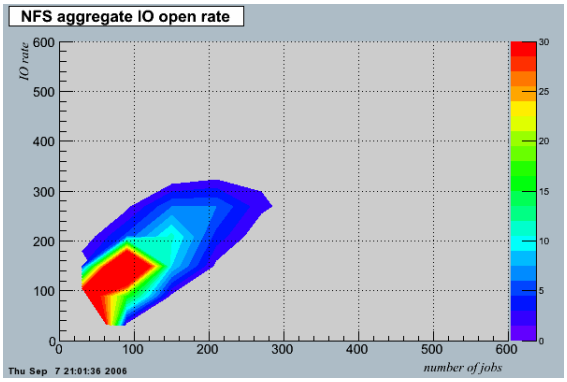
$$(s_{job} < T_i \bigcap e_{job} > T_j)$$

For representation of all test, the contour plots has been chosen, where x-axis is number of jobs and y-axis is aggregated IO related to particular number of jobs. It gives not only a opportunity to see most frequent values which has been measured and seen, but also it helps to somehow reflect non-dedicated environment for measurement. In previous chapters, we mentioned that each request (not previously presented at cache) in xrootd needs to be delayed for fixed time. This time can actually slow the performance comparing to other systems. Therefore, we were interested to see 2 different cases:

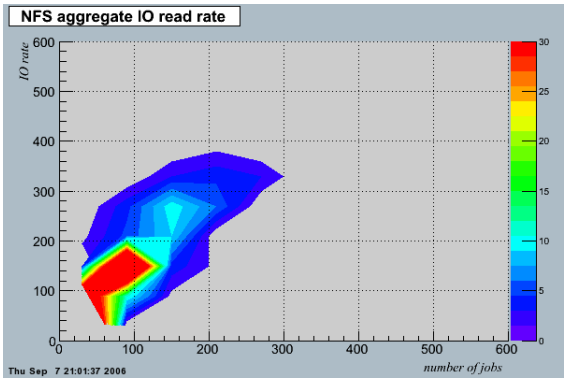
- **read rate** - measured IO rate without open and close delays
- **open rate** - measured IO rate with open and close delays

The other storage solutions available within STAR framework, which we could measured were: *Rootd* and *Panasas* exposed to users via NFS protocol.

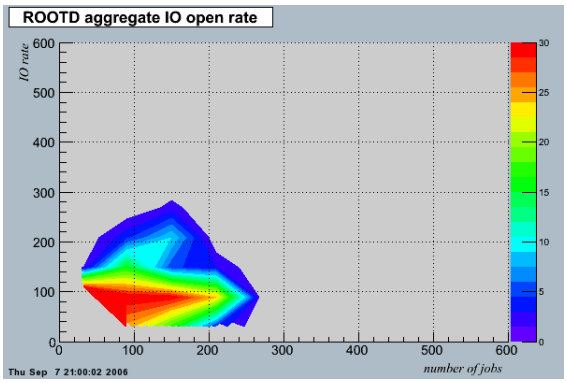
The figure 3.16 shows results from all run tests on all mentioned storage solutions. These results show that xrootd scales with number of jobs as the best compare to all other solutions and even has most values placed higher than others. This signs that even commercial and very expensive solution (Panasas) with its storage area network (SAN) model has poorer results than tenfold cheaper solution (xrootd) with its Direct Attached Storage (DAS) model.
Moreover, it is observable, that xrootd with its fixed delay time has still better performance comparing to other solutions.



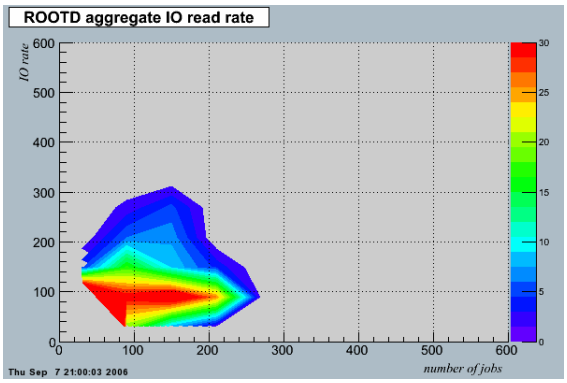
(a) NFS aggregate IO open rate



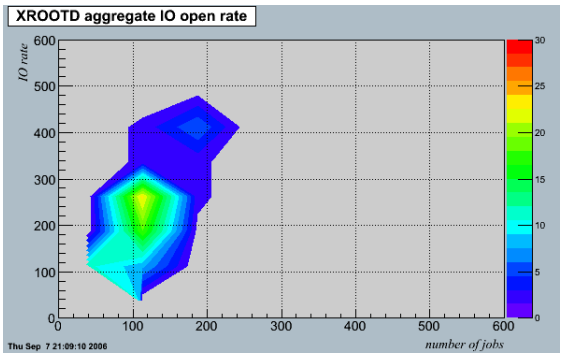
(b) NFS aggregate IO read rate



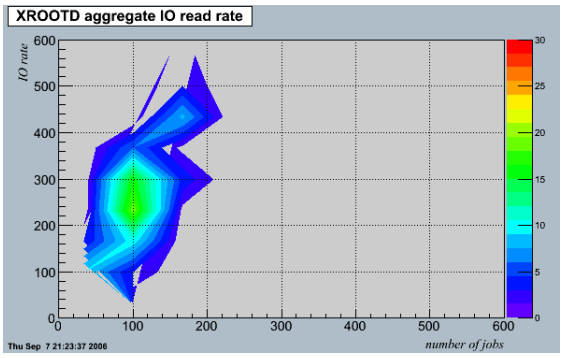
(c) Rootd aggregate IO open rate



(d) Rootd aggregate IO read rate



(e) Xrootd aggregate IO read rate



(f) Xrootd aggregate IO read rate

Figure 3.16: Aggregate IO comparison of several storage solutions

Chapter 4

Improving Xrootd

While the Xrootd seems to perform extremely well and satisfy STAR's most immediate needs, such as a storage solution serving high-performance, scalable, fault-tolerant access to their physics data, it could itself be improved and extended. For example, Xrootd does not move files from one data-server to other data-server or even from one cache to other cache within one node, but always restore files from MSS. This may be slow and inefficient in comparison with transferring the file from other node or cache, not involving any tape mount or other delays intrinsic to MSS. Additionally, the system is not able import files from other space management systems (as dCache, Castor [52]) or even across the grid. In a large scale pool of nodes, if "*ALL*" clients ask for a file restore from MSS, the system would exhibit a lack of coordination of accesses MSS resources as it lacks a request queue. This advanced feature is needed for any coordinated requests and is especially important in a shared access environment where other tools, such as bulk data transfers to remote sites, may also perform MSS staging requests. There are no advanced reservations of space, other users can collate the space in the meantime while the restore from MSS operation is still ongoing (in fact, we have observed and reported in section 3.4 failures related to the lack of space, likely related to such timing issues). There are no extended policies per users or role based giving *advanced* granting of permissions to a user. There is no concept of pinning the files, requested files can be evicted to release a space. This makes un-practical additional features such a pre-staging (essential for efficient co-scheduling of storage and computing cycles).

In addition, there are other middle-ware designed for the space management and only for the space management. Specifically, the grid middle-ware component called **Storage Resource Managers (SRMs)** [53], [54], [55] has for function to provide dynamic space allocation and file management on shared distributed storage systems. SRMs are designed to manage space, meaning designed to negotiate and handle the assignment of space for users and also manage lifetime of spaces. In addition of file management, they are responsible for managing files on behalf of user and provide advanced features such as pinning files in storage till they are released or also even manage lifetime of files that could be removed after specific time. SRMs also manage file sharing with configurable policies regulating what should reside on storage or what to evict. One of the powerful features of SRMs is ability of bringing the files from other SRMs, local or at remote locations including from other site and across the Grid . In fact, SRMs defines a fully specified protocol aims to handle and negotiate requests and movements. Note that SRMs themselves do not move files: they negotiate space and orchestrate file movements using standard transfer tools (ssh, gsiftp, bbftp for example) and it keeps a track of transfers and recover them from failures.

SRMs comes in three flavors of storage resource managers:

- **Disk Resource Manager (DRM)**
  - manages one or more disk resources
- **Tape Resource Manager (DRM)**
  - manages the tertiary storage system (e.g. HPSS)

- **Hierarchical Resource Manager (HRM=TRM+DRM)**
  - stages files from tertiary storage into its disk cache and manage both resources

On the other hand, while SRMs do manage space efficiently and can talk to other SRM (bringing for example files from other caches or SRM-aware tools), they know nothing of load balancing capabilities and they do not perform data aggregation or provide any global view of storage space, all of which was showed as a key advantage of Xrootd. We therefore proposed to leverage these technologies and integrate to Xrootd and SRM back-end for managing space.

4.1 XROOTD - SRM architecture design

Both systems have their own inner architecture and the task of integration lies on the question on how to bind them together. Fortunately, Xrootd with its flexible layered architecture and interfaces [36] allows us to easily replace unwanted ones by another implementation incorporating SRM protocol has it is showed in figure 4.1. Xrootd then remains responsible for managing disk cluster and the access to the

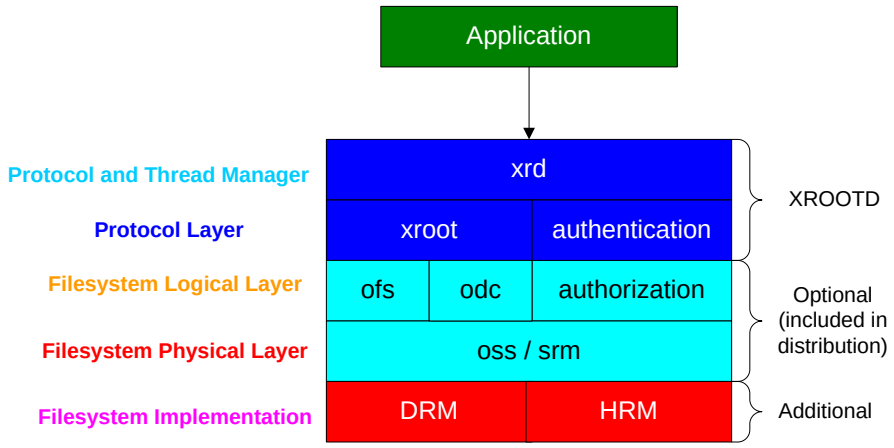


Figure 4.1: XROOTD - SRM components interaction

global name-space, DRM will be accountable for managing disk cache and HRM will be responsible for staging files from MSS. In other words xrootd becomes a client of SRMs.

While there still remains a lot to discuss, a first design and the implementation of xrootd server interaction with HRM has been done. Before further description, we have to define terms being used in SRM terminology with knowledge of previously defined LFN/PFN:

- **File names** (forms of PFN, but have here additional dimension to satisfy the SRM needs of an explanation)

  - **SFN** - a file name assigned by a site to a file. Normally, the site file name will consist of a "machine:port/directory/file"
  - **TFN** - the "*transfer*" file name of the actual physical location of a file that needs to be transferred. It has a format similar to an SFN.

- **URL** - represents a location of a file on the grid
  - **SURL** - a "*site*" URL which consists of "*protocol://SFN*". The protocol can vary, since the SRM supports protocol negotiation. For communication with srm, the protocol is srm, other variant is for example gridftp.
  - **TURL** - a "*transfer*" URL that an SRM returns to a client for the client to "*get*" or "*put*" a file in that location. It consists of "*protocol://TFN*", where the protocol must be a specific transfer protocol selected by SRM from the list of protocols provided by the client

Conceptually, since the HRM is a server, the xrootd server need to understand and make the calls to the API of HRM client exposed by the CORBA technology.
The calls have the following semantic:

- **srmGet** - given a source SURL (site-URL), an srmGet will get a file from the source site, pins the file for a lifetime (a config file parameter) and return a TURL (transfer-URL). If the file is already in cache, it simply pins the file, and returns a TURL. It is expected that after the file is used by the xrootd client, it will be followed with an srmRelease.

- **srmPut** - an srmPut is for the purpose of pushing a file into the cache. The DRM will allocate space, and return a TURL where the client can put the file. It is possible to specify in the call a target SURL which is the final destination of the file. It is expected that after the file was put into the disk by the xrootd client, it will be followed with an srmPutDone.

- **srmCopy** - given a file in the disk cache, an srmCopy will copy the file to a remote location using a target SURL. With xrootd, this can be used for a request to archive a file into MSS, that is in the disk cache.

- **srmModify** - this is a request to change the content of an existing file. The request may be to modify the existing content or append to it. The file is expected to be in the cache. If it is not found in the cache an error is returned. To get a file that is not in the cache, an srmGet should precede the srmModify call. The DRM will allocate extra space if size is specified, or assign the maximum default allowed. Similar to an srmPut, it is possible to specify in the call a target SURL which is the final destination of the file. It is expected that after the modification to the file was finished, it will be followed with an srmPutDone.

- **srmRemove** its funcition is to remove a remote file from the MSS. Thus, a delete request by the client will generate an srmRemove with the target SURL. To remove the file in the disk cache only an srmRemove with the LFN should be provided. To remove from disk cache and the archive, two srmRemove calls should be made, one with LFN as a parameter, and one with the target SURL

All mentioned calls can xrootd use for reading a file from MSS, writing a file into MSS or modifying the files in MSS. The table 4.1 shows the summary of all available operation and forms of the particular SRM calls.

Table 4.1: Summary of all functions available to xrootd

Type of operation	**Form of the SRM call**
Copying into cache	srmPut(FID=LFN, source=null, target=SURL)
Remove from cache	srmRemove(FID=LFN)
Remove from MSS	srmRemove(FID=SURL)
Copying into MSS	srmCopy(FID=LFN, source=null, target=SURL)
Modifying without archive	srmModify(FID=LFN,source=null,target=null)
Modifying with archive	srmModify(FID=LFN,source=null,target=SURL)
Release a file	srmRelease(requestID,FID=SURL)
Finish putting to cache	srmPutDone(requestID,FID=SURL)
Finish modifying in cache	srmPutDone(requestID,FID=SURL)
Abort request	srmAbort(requestID)
Status of an operation	srmStatus(requestID,FID=SURL)

On xrootd side, the *oss component* has been extended as a generic plugin with virtual C/C++ functions. It offers a possibility to create a new plugin calling the mentioned SRM calls within HRM API written in C++.
This initial approach leads to have a large centralized cache visible to all nodes and managed by HRM. We have successfuly deployed the HRM server in front of the BNL HPSS and tested all mentioned calls implemented in new xrootd plugin by reading and writing files from/to HPSS.

The next work would be focused on a capability of transferring files from HRM cache to the DRM deployed on each node of the farm. Therefore, to have an ability of the dynamically populated distributed disk using HRM (HPSS), managed by DRM (disk caches) and controlled by xrootd.

Chapter 5

Conclusion

In this work we have studied some key aspects of architectures, topologies, technologies, methods and algorithms aimed to fulfill the task of managing and accessing very big amounts of data. Our studied case was carried within the RHIC/STAR experimental and analysis environment, one of the biggest High Energy and Nuclear Physics (HENP) experiments producing PetaByte of data per year.

We started with a short description of the STAR experiment, continued with its expectation of data sizes to the future and the available computing resources which experiment can use. We discussed different topologies (centralized, distributed) of data storage in the highlight of the performance clusters as well as differences, advantages, disadvantages of hardware and software solutions with application to real examples of the storage solutions.

While STAR has chosen a distributed topology as their primary storage solution many years ago, we discussed their current data model and showed its bottlenecks leading to the search for other (better) distributed storage solution. We have discussed two diverse distributed solutions (dCache [31] and Xrootd [36]) well-known in HENP environment. Our discussion was focused on the architectures of these two systems with aspect to performance, scalability and name-space approaches. The result of this discussion evolved toward the main work of this report. It is a performance and scalability evaluation of xrootd system in very large scale.

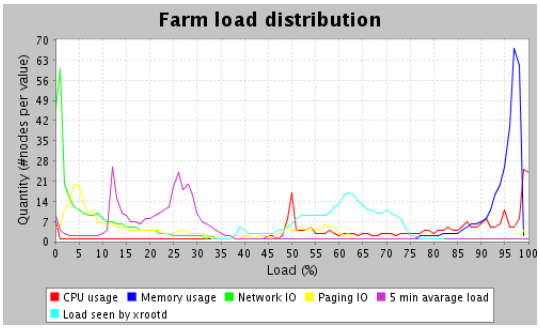
For the achievement of this goal, we started with basic deployment of the system on the 300 nodes, followed by enabling and stabilization of the access to the tape system. Big portion of the work was dedicated to approaches of tuning the IO, such as load balancing in distributed environment. We showed how to achieve ideal load distribution of the system and therefore increase the IO throughput. The scalability of the system has been proved by integrating the system into STAR framework and monitor its behavior under many user's requests.

To discover the question of high-performance, we made a measurement of aggregate IO of all available storage solutions within STAR framework. We have observed from the results, that xrootd performs best comparing to other solutions, even to expensive centralized storage solution represented by Panasas [14]. Therefore, we have showed that XROOTD is excellent high-performance and scalable solution for serving large amount of data.

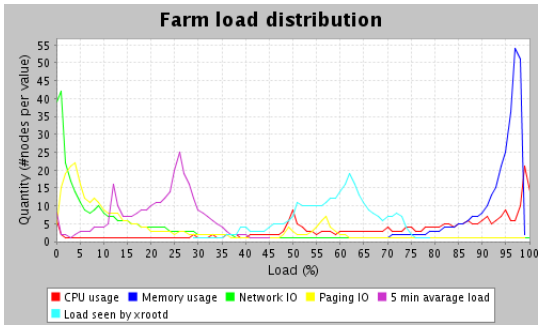
In the last chapter, we have presented our improvements to Xrootd in order to have an access to Grid and therefore having capability of files spread on other sites or clusters.

Appendix A

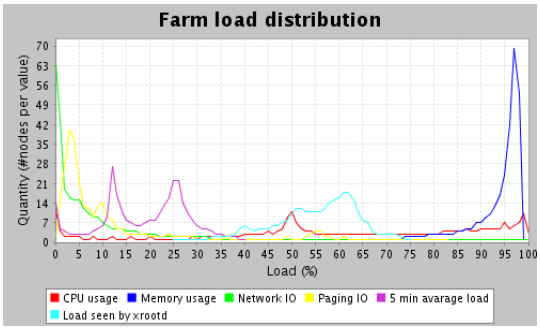
Load statistic



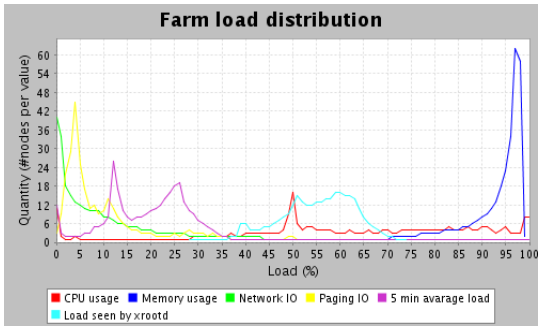
(a) Week 22



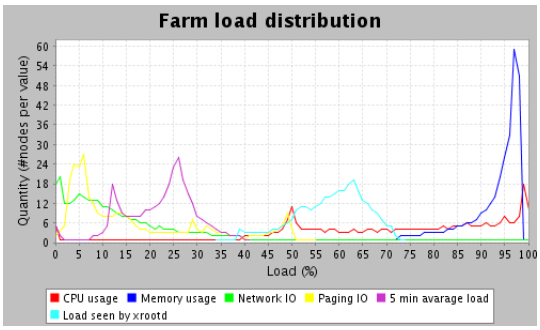
(b) Week 23



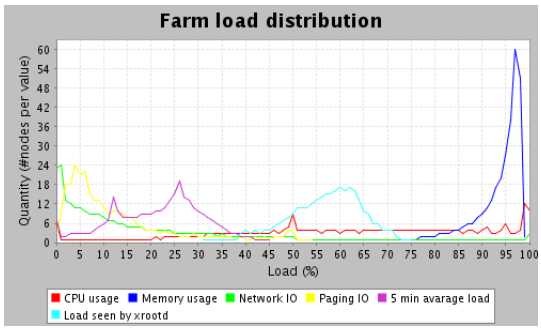
(c) Week 24



(d) Week 25

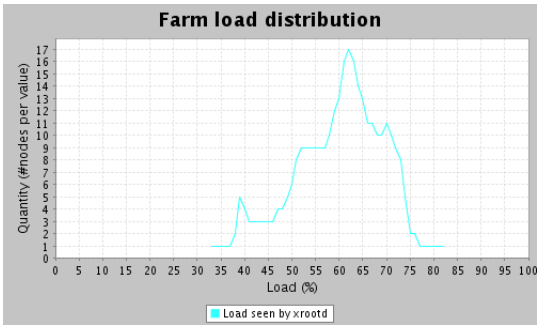


(e) Week 26

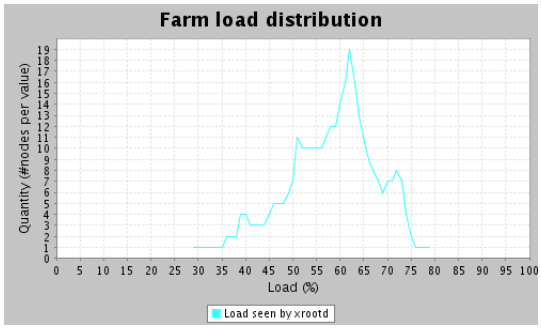


(f) Week 27

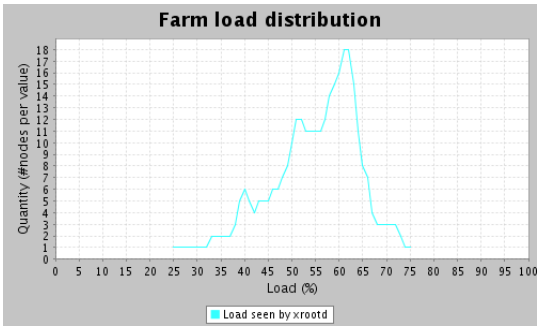
Figure A.1: **Summary of all individual load factors gathered on the *STAR CAS cluster*** (cluster dedicated for analysis jobs)



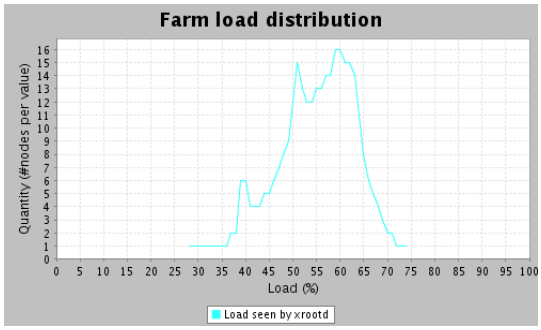
(a) Week 22



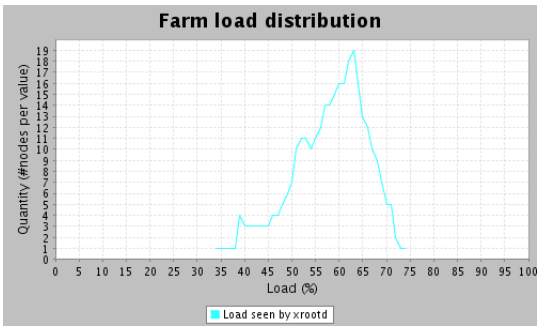
(b) Week 23



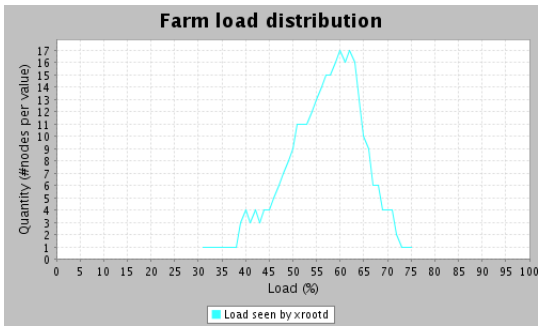
(c) Week 24



(d) Week 25

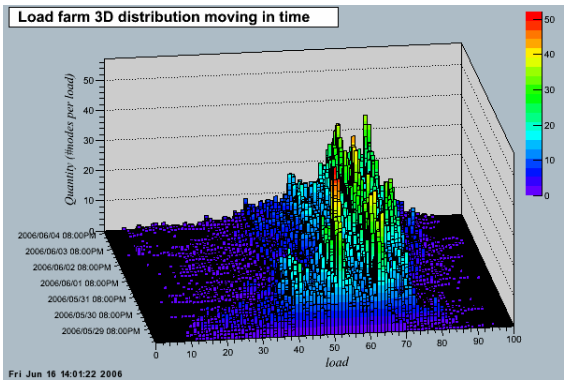


(e) Week 26

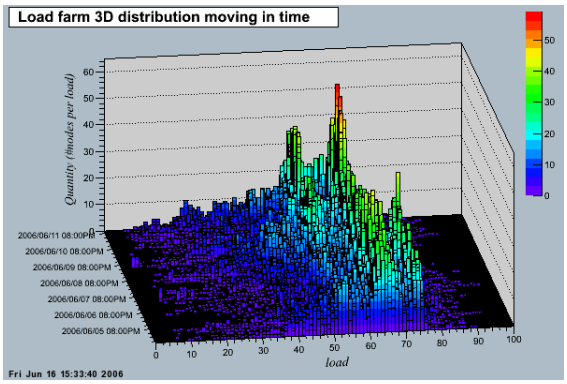


(f) Week 27

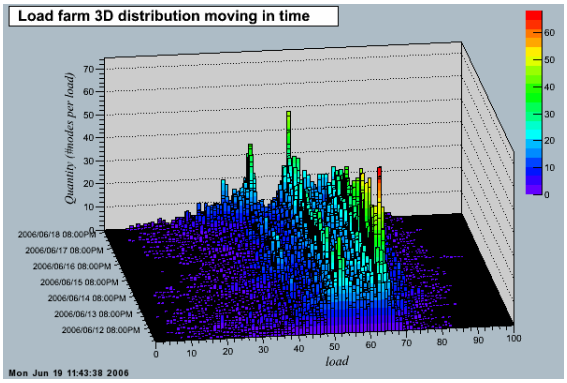
Figure A.2: **Xrootd load distribution statistic on the *STAR CAS cluster*** (cluster dedicated for analysis jobs showing very stable distribution over many weeks)



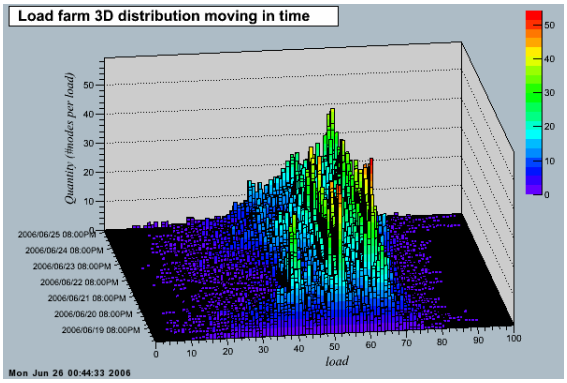
(a) Week 22



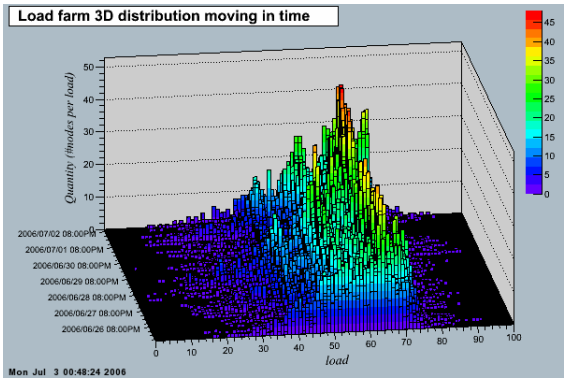
(b) Week 23



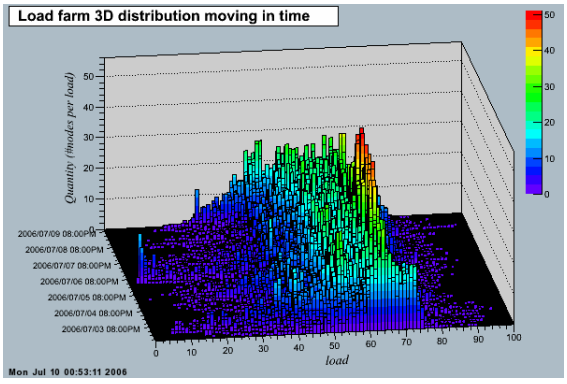
(c) Week 24



(d) Week 25

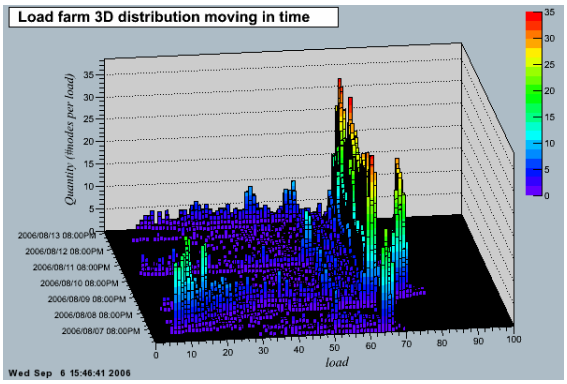


(e) Week 26

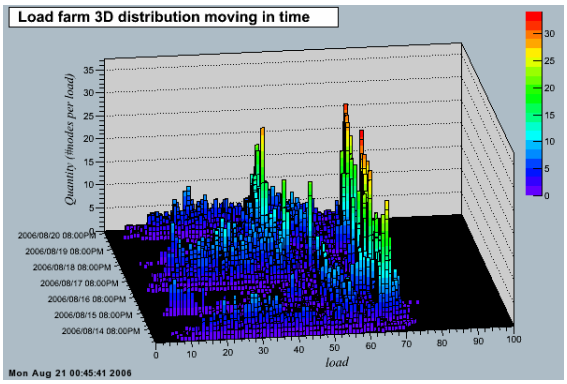


(f) Week 27

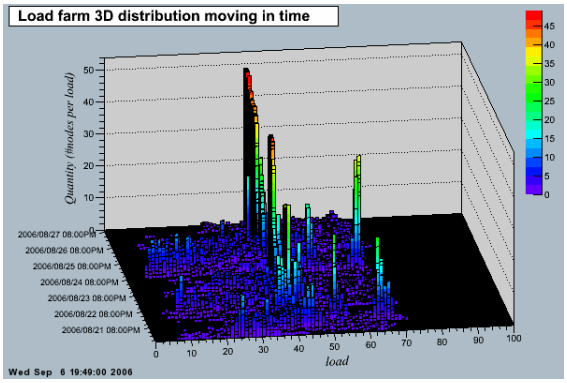
Figure A.3: **3-D Xrootd load distribution statistic with time dependency on the *STAR CAS cluster*** (cluster dedicated for analysis jobs showing very stable load distribution over many weeks)



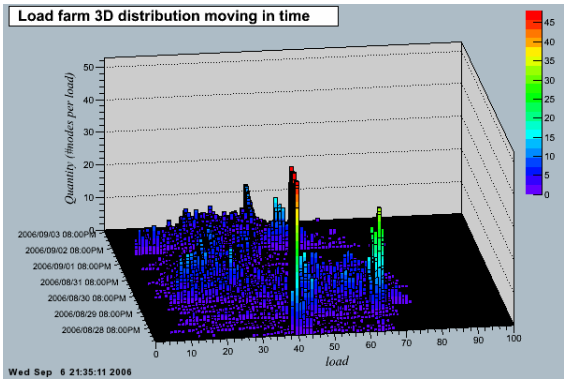
(a) Week 32



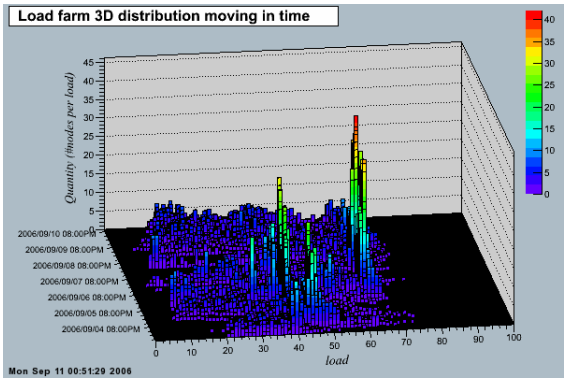
(b) Week 33



(c) Week 34



(d) Week 35



(e) Week 36

Figure A.4: **3-D Xrootd load distribution statistic with time dependency on the *STAR CRS cluster*** (cluster dedicated for reconstruction jobs showing lots of fluctuations introduced by the identical behavior of reconstruction's jobs)

Bibliography

[1] STAR experiment. [Online]. Available: http://www.star.bnl.gov

[2] Relativistic Heavy Ion Collider. [Online]. Available: http://www.bnl.gov/RHIC

[3] Brookhaven National Laboratory. [Online]. Available: http://www.bnl.gov

[4] Quark Gluon Plasma. [Online]. Available: http://en.wikipedia.org/wiki/Quark_gluon_plasma

[5] T. Christiansen and N. Torkington, *Perl Cookbook*. O'Reilly & Associates, Inc., 1998.

[6] L. Wall and R. L. Schwartz, *Programming perl*. O'Reilly & Associates, Inc., 1994.

[7] J. Vromans, *Perl (Pocket reference)*. O'Reilly & Associates, Inc., 2002.

[8] P. DuBois, *MySQL*. Sams Publishing, 2005.

[9] A. Descartes and T. Bunce, *Programming the perl DBI*. O'Reilly & Associates, Inc., 2000.

[10] A. Inc., "A storage architecture guide," STORAGEsearch.com, May 2000.

[11] D. Alabi, "NAS, DAS, SAN ? - choosing the right storage technology for your organization," STORAGEsearch.com, May 2004.

[12] D. Patterson, G. Gibson, and R. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *Proc. Int'l Conf. Management of Data*. ACM, 1989, pp. 109–116.

[13] J. May, *Parallel I/O for High Performance Computing*. Academic press, 2001.

[14] D. Nagle, D. Serenyi, and A. Matthews, "The panasas activescale storage cluster - delivering scalable high bandwidth storage," in *Proc. of the ACM/IEEE SC2004*, November 2004.

[15] Panasas file system (panfs). [Online]. Available: http://www.panasas.com/panfs.html

[16] P. Schwan, "Lustre: Building a file system for 1000-node clusters," 2003. [Online]. Available: citeseer.ist.psu.edu/schwan03lustre.html

[17] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proc. of the First Conference on File and Storage Technologies (FAST)*, Jan. 2002, pp. 231–244. [Online]. Available: citeseer.ist.psu.edu/schmuck02gpfs.html

[18] S. Ghemawat, H. Gobioff, and S. Leung, "The google file system," 2003. [Online]. Available: citeseer.ist.psu.edu/ghemawat04google.html

[19] D. Roselli, J. R. Lorch, and T. E. Anderson, "A comparison of file system workloads," pp. 41–54. [Online]. Available: citeseer.ist.psu.edu/roselli00comparison.html

[20] P.Jakl, *et al.*, "From rootd to xrootd, from physical to logical files: experience on accessing and managing distributed data," in *Proc. of Computing in High energy and nucler physics (CHEP'06)*, 2006.

[21] Rootd. [Online]. Available: http://root.cern.ch/root/NetFile.html

[22] R. Brun and F. Rademakers, "Root - an object oriented data analysis framework," in *Proceedings AIHENP'96 Workshop, Lausanne.* Nucl. Inst. & Meth. in Phys. Res. A 389 (1997), Sep. 1996, pp. 81–86.

[23] High Performance Storage System. [Online]. Available: http://www.hpss-collaboration.org/hpss/index.jsp

[24] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic metadata management for petabyte-scale file systems." [Online]. Available: citeseer.ist.psu.edu/698264.html

[25] STAR computing. [Online]. Available: http://www.star.bnl.gov/STAR/comp/

[26] G. Singh, *et al.*, "A metadata catalog service for data intensive applications," 2003. [Online]. Available: citeseer.ist.psu.edu/singh03metadata.html

[27] V. Mandapaka, C. Pruneau, J. Lauret, and S. Zeadally, "Star-scheduler: A batch job scheduler for distributed i/o intensive applications," 2004. [Online]. Available: http://www.citebase.org/abstract?id=oai:arXiv.org:nucl-ex/0401032

[28] T. E. Anderson, *et al.*, "Serverless network file systems," in *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles.* New York, NY, USA: ACM Press, 1995, pp. 109–126.

[29] E. Levy and A. Silberschatz, "Distributed file systems: concepts and examples," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 321–374, 1990.

[30] C. Kesselman and I. Foster, *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, November 1998. [Online]. Available: http://www.amazon.fr/exec/obidos/ASIN/1558604758/citeulike04-21

[31] P. Fuhrmann, "dCache, the commodity cache," in *Proc. of Twelfth NASA Goddard and Twenty First IEEE Conference on Mass Storage Systems and Technologies*, 2004.

[32] A. Dorigo, P. Elmer, F. Furano, and A. Hanushevsky, "Xrootd - a highly scalable architecture for data access," in *Proc. WSEAS'05*, 2005.

[33] P. Fuhrmann, "dCache, a distributed data storage caching system," in *Proc. of Computing in High energy and nucler physics (CHEP)*, 2001.

[34] P. Fuhrmann, "dCache, lcg se and enhanced use cases," in *Proc. of Computing in High energy and nucler physics (CHEP)*, 2004.

[35] P. Fuhrmann, "dCache, the overview," White paper, 2004. [Online]. Available: http://www.dcache.org

[36] A. Hanushevsky, A. Dorigo, and F. Furano, "The next generation root file server," in *Proc. CHEP'04*, 2004.

[37] A. Hanushevsky and H. Stockinger, "Proxy service for the xrootd data server," in *Proc. SAG'04*, 2004.

[38] Deutsches elektronen-synchrotron (DESY). [Online]. Available: http://www.desy.de

[39] Fermi national accelerator laboratory (fermilab). [Online]. Available: http://www.fnal.gov/

[40] ROOT framework. [Online]. Available: http://root.cern.ch

[41] F. Furano, "Large scale data access: Architectures and performance," Ph.D. dissertation, University of Venezia, Department of Informatics, January 2006.

[42] A. Hanushevsky and B. Weeks, "Scalla: Scalable cluster architecture for low latency access, using xrootd and olbd servers," White paper, 2006. [Online]. Available: http://xrootd.slac.stanford.edu/papers/Scalla-Intro.htm

[43] S. Androutsellis-Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 335–371, 2004.

[44] A. Hanushevsky, *XRD Configuration Reference*, SLAC, 2005. [Online]. Available: http://xrootd.slac.stanford.edu/

[45] A. Hanushevsky, *Open File System and Open Storage System Configuration Reference*, SLAC, 2005. [Online]. Available: http://xrootd.slac.stanford.edu/

[46] A. Hanushevsky and G. Ganis, *Authentication and Access Control Configuration Reference*, SLAC, 2005. [Online]. Available: http://xrootd.slac.stanford.edu/

[47] A. Hanushevsky, *Open Load Balancing Configuration Reference*, SLAC, 2005. [Online]. Available: http://xrootd.slac.stanford.edu/

[48] G. Anderson and P. Anderson, *The unix C shell field guide.* Prentice-Hall, 1986.

[49] A. Hanushevsky, *Cache File System Support MPS Reference*, SLAC, 2004. [Online]. Available: http://xrootd.slac.stanford.edu/

[50] A. Hac and T. Johnson, "A study of dynamic load balancing in a distributed system," in *SIGCOMM '86: Proceedings of the ACM SIGCOMM conference on Communications architectures & protocols.* New York, NY, USA: ACM Press, 1986, pp. 348–356.

[51] K. Bubendorfer and J. H. Hine, "A compositional classification for load-balancing algorithms," Victoria University of Wellington, Tech. Rep. CS-TR-99-9, July 1998.

[52] O. Bcarring, *et al.*, "Storage resource sharing with CASTOR," in *In Proceedings of the 12th NASA Goddard*, vol. 21st IEEE Conference on Mass Storage Systems and Technologies, April 2004, pp. 345–359.

[53] A. Shoshani, A. Sim, and J. Gu, *Storage Resource Managers: Essential Components for the Grid.* Kluwer Academic Publishers, 2003, ch. In Grid Resource Management: State of the Art and Future Trends, pp. 321–340.

[54] L. Bernardo, A. Shoshani, A. Sim, and H. Nordberg, "Access coordination of tertiary storage for high energy physics applications," in *IEEE Symposium on Mass Storage Systems*, 2000, pp. 105–118. [Online]. Available: citeseer.ist.psu.edu/bernardo00access.html

[55] A. Shoshani, A. Sim, and J. Gu, "Storage resource managers: Middleware components for grid storage," 2002. [Online]. Available: citeseer.ist.psu.edu/shoshani02storage.html