ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta jaderná a fyzikálně inženýrská

Katedra matematiky

DISTRIBUOVANÁ SPRÁVA DAT V EXPERIMENTU STAR

Rešeršní práce

Vypracoval: Pavel Jakl
Vedoucí práce: Michal Šumbera CSc.
Konzultant: Dr. Jérôme Lauret, Brookhaven National Laboratory, USA
Školní rok: 2004/2005

Contents

Chapter 1

Introduction

Driven by increasingly complex problems and propelled by increasingly powerful technology, today's science is as much based on computation, data analysis, and collaboration as on the 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.

Ten years ago, biologists were happy to compute a single molecular structure. Nowadays, they want to calculate the structures of complex assemblies of macromolecules and screen thousands of drug candidates. Personal computers now ship with up to 100 gigabytes (GB) of storage. But several High Energy Physics projects such as RHIC (Relativistic Ion Heavy Collider) , is produced 5 TB[1] of data per year or the bigger project Large Hadron Collider will produce 15 TB of data per year.

The grid as a distributed system allows scientists to access computer power and data from around the world seamlessly, without needing to know where the computers are situated.
Grid is a new class of parallel and distributed system. It provides scalable, secure, high-performance mechanisms for discovering and negotiating access to remote resources. Analysis for particle physics can also be done on conventional supercomputers, but these are expensive and in high demand. The grid, in contrast, is constructed from thousands of cheap units that can be increased to meet users' needs.

In this report I will try to point distributed data management in the sense of experiment STAR. I will describe tools to run physics particle's analysis jobs and essential part is dedicated to system XROOTD which was developed to achieve the goal of the distributed data management.

[1]TB = terabyte = 10^{12} bytes

Chapter 2

The STAR experiment

2.1 What is STAR and its goal

The Relativistic Heavy Ion Collider (RHIC) at the Brookhaven National Laboratory is currently the world's highest energy accelerator of heavy nuclei and the world's first polarized proton collider.

The Solenoidal Tracker at RHIC (STAR)is the detector located at the 6 o'clock position at the RHIC collider ring. STAR is designed to study the behavior of strongly interacting matter at high energy density and to search for signatures of Quark Gluon Plasma (QGP) formation.

The STAR detector produces raw data of particle tracks and stores them into HPSS (High Performance Storage System) in form of .daq files. After that, it is ran a reconstruction process over daq files and grow up the structure called "event". This event is stored in shape of MuDST file.

MuDST is a structured file which is further used for purpose of analysis of some particle tracks. The analysis of some tracks is made with help of application ROOT which is an object- oriented data analysis framework.

All of those analyses and reconstructions are executed on big computing facility called RCF (RHIC Computing Facility) with over 380 Dual Pentium III or IV computers.

2.2 Tools to handle the analysis

In this section we will pursue with description of tools that are used during the analysis.

2.2.1 Batch queuing systems

The typical batch queuing system schedules jobs for execution using a set of queue controls. Within each queue, jobs are then selected in first-in, first-out (FIFO) order.
The purpose of BQS (Batch queuing systems) is to provide additional controls over

initiating or scheduling execution of batch jobs[1], and to allow routing of those jobs between different hosts of a farm.
Each BQS allows to the system administrator to define what types of resources, and how much of each resource, can be used by each job. It has a full knowledge of the available queued jobs, running jobs, and system resource usage. According to some configurable scheduling policies the job is sent and enqueues to a queue.

Some of the most popular BQS are:

Condor targeted at using idle workstations

NQS basic functionality

LSF (Load Sharing Facility) most popular, very rich set of features, otherwise licensed software

PBS (Portable batch system)

SGE (Sun grid engine) very popular batch system

2.2.2 File Catalog

The FileCatalog database contains information about all files used in production for the STAR experiment.
This database holds all information about files which are populated in STAR experiment.

The FileCatalog perl module is intended to provide access to the databse both for data querying and retrieval as well as data insertion and modifications. It is based on the concept of setting keywords within a context persistency. The user first sets a context using to a set of keywords with the desired value (or conditions), and then uses special commands to get/insert/delete/modify the data in the database.

The perl utility is called get_file_list.pl and it is availably from each node at the farm.

2.2.3 STAR Unified Meta Scheduler

The STAR scheduler was development to allow users to submit a job on several input files, residing on NFS or on the various local disks of the farm machines. The scheduler will divide the job into different processes that will be dispatched to different machines using the Batch queuing systems. In order for the scheduler to do that, the user has to follow some simple rules when writing his program, and to write an XML file describing the job he wants to be done.

A job description can look like this:

[1]meaning the invocation of an appropriate command interpreter (e.g. /bin/csh, /bin/sh)

```
<?xml version="1.0" encoding="utf-8" ?>
<job>
  <command>
      root4star -q -b StRoot/StHbtMaker/doc/StHbtDiHadron.C("\$FILELIST")
  </command>
  <stdout URL="file:/star/u/pjakl/scheduler/out/\$JOBID.out"/>
  <input URL="file:/star/data21/.../st_physics_2312011_raw_0017.MuDst.root"/>
  <input URL="file:/star/data21/.../st_physics_2312011_raw_0016.MuDst.root"/>
  <input URL="file:/star/data21/.../st_physics_2312011_raw_0015.MuDst.root"/>
</job>
```

The user just specifies all of his requirements of the job in XML-form and submit his job by typing the command star-submit jobDescription.xml and job is dived into processes and enqueues to a queue.

2.2.4 High performance storage system

High performance storage system(HPSS) is software that manages hundreds of terabytes to petabytes of data on disk and robotic tape libraries. HPSS provides highly flexible and scalable hierarchical storage management that keeps recently used data on disk and less recently used data on tape. HPSS uses cluster and SAN technology[2] to aggregate the capacity and performance of many computers, disks, and tape drives into a single virtual file system of exceptional size and versatility.

This approach enables HPSS to easily meet otherwise unachievable demands of total storage capacity, file sizes, data rates, and number of objects stored. HPSS provides a variety of user and filesystem interfaces ranging from the ubiquitous vfs, ftp, samba and nfs to higher performance pftp, client API, local file mover and third party SAN.

There are 4 main features why to use a HPSS:

- **Scalable Capacity** As architects continue to exploit hierarchical storage systems to scale critical data stores beyond a petabyte (1024 terabytes) towards an exabyte (1024 petabytes), there is a equally critical need to deploy a high performance, reliable and scalable HSM[3].
- **Scalable I/O Performance** As processing capacity grows from trillions of operations per second towards a quadrillion operations per second and data ingest rates grows from 10s of terabytes per day to 100s of terabytes per day, HPSS provides a scalable data store with reliability and performance to sustain 24x7 operations in demanding high availability environments
- **Incremental Growth** - As data stores with 100s of terabytes and petabytes of data become increasing common place, HPSS provides a reliable and flexible solution

[2]a Storage Area Network (SAN) is a network designed to attach computer storage devices such as disk array controllers and tape libraries to servers

[3]Hierarchical Storage Management is a data storage system that automatically moves data between high-cost and low-cost storage media

that scales seamlessly using heterogeneous storage devices, robotic tape libraries, and processors connected via LAN, WAN and SAN.

- **Reliability** As data stores increase in from 10s of million of files to 100s of millions of files and collective storage I/O rates grow to 100s of terabyte per day, HPSS provides a scalable metadata engine based on IBM's DB2 to ensures highly reliable and recoverable transactions down to the I/O block level.

2.2.5 NFS Area

NFS stands for Network File System, a file system developed by Sun Microsystems, Inc. It is a client/server system that allows users to access files across a network and treat them as if they resided in a local file directory. For example, if you were using a computer linked to a second computer via NFS, you could access files on the second computer as if they resided in a directory on the first computer. This is accomplished through the processes of exporting (the process by which an NFS server provides remote clients with access to its files) and mounting (the process by which file systems are made available to the operating system and the user).

The NFS protocol is designed to be independent of the computer, operating system, network architecture, and transport protocol. This means that systems using the NFS service may be manufactured by different vendors, use different operating systems, and be connected to networks with different architectures. These differences are transparent to the NFS application, and thus, the user.

2.2.6 AFS Area

The Andrew File System (AFS) Service is a storage service offered by UITS to those faculty, staff, and graduate students at Indiana University who participate in inter-institutional collaborations that have chosen AFS as their primary means of data sharing.

AFS is a global file system. AFS directories worldwide become available to a user with access to a host with an AFS client installed. It is currently in widespread use in academia and in scientific labs around the world. Each AFS site has its own AFS cell, which is usually named after the institution or a research group that administers it.

For example, Brookhaven lab's AFS cell is called **rhic.bnl.gov**, CERN's AFS cell is called cern.ch, NASA's AFS cell is called jpl.nasa.gov, etc. Indiana University's AFS cell name is ovpit.indiana.edu.

2.3 Data oriented computing

The STAR experiment at the Relativistic Heavy Ion Collider produces a huge amount of data to be accessed by a high number of analysis jobs. For this reason it requires a reliable and scalable data access system.

XROOTD system is based on the persistency mechanism of the C++ ROOT framework, developed at CERN, that is able to stream an object on a binary file in a similar way as the Java framework does. ROOT also provides a remote file access mechanism via a TCP/IP-based data server daemon known as **rootd** which has the only purpose to serve opaque data. A plugin manager (fig. 2.1) hides the user from the actual location of the files, by masking the path (local file system or remote server) which is going to use to access the data.

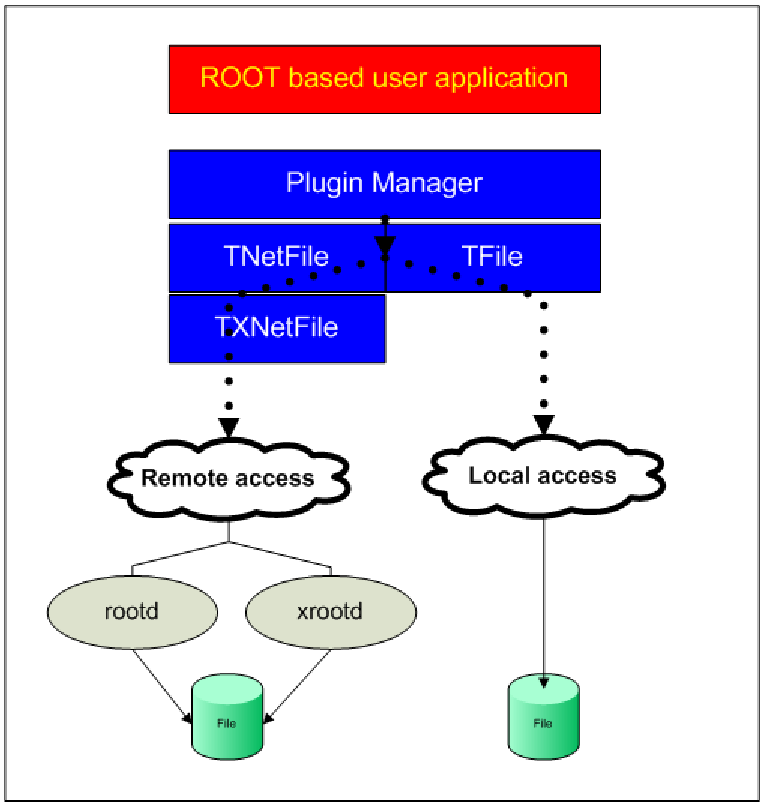

Figure 2.1: Transparent local and remote access

User doesn't know anything about reading a file locally or remotely and even the using of TNetFile instance (rootd) or XTNetFile (xrootd).

Remote access to the files can be achieved by the other remote access mechanism, such for example NFS. A deeper requirements analysis shows that this kind of solution is not acceptable for many reasons:

- **the size of the data repository**: thousands of thousands of files must be scattered among multiple servers. Beside the complex organization needed, NFS does not provide any tool useful to automatically manage the choice of the right mounted partition disk to access for a request, i.e. what is often called "logical to physical name resolution"

- **number of concurrent accesses to the data**: thousands of concurrent accesses from end users and batch jobs, that continuously analyze the data in a completely random way, greatly overcome the scalability of the basic NFS architecture
- **software engineering issues**: NFS simulates a local file system and most jobs are not much tolerant to all possible troubles accessing local files
- **administered issue**: the use of NFS would implies mounting remote volumes on the machine where the user is running his jobs; this is not acceptable from the point of view of many system administrators, and especially in the case in which the user machine is a desktop computer or even a laptop
- **different design of NFS**: if for any reason some NFS servers are having troubles, typically the machine mounting the remote volumes will experience problems when any NFS access is tried, with no user (or application) control of timeouts, retries etc.

To overcome some of these limitations, the alternative of building a data server suggests a different paradigm which can be deployed or extended in order to satisfy the heavy requirements of the data analysis tasks. At the server side, rootd offers the solution to share this big load between many machines keeping the files on their local disks, while at the client side, a specialization of the ROOT's data access classes can provide a way to access the remote data which is transparent to the users of the framework.

The deployment of big processing farms, as well as of data access systems able to handle millions of scattered files must be able to give data processing services to a wide community of users with high availability and performances.
Some of the needed characteristics are:

- multiple servers have to cooperate with the purpose of handling huge amounts of distributed (and redundant if necessary) data without forcing the client to know which server to contact to access a particular file
- the server has to hide the client applications from its underlying file system types (meaning one of the Mass Storage Systems[4])
- a load balancing mechanism is needed, in order to efficiently distribute the load between clusters of servers
- the system resources (sockets, memory, cache, disk accesses, cpu cycles, etc.) have to be used at the best, at both client and server sides
- a high degree of fault tolerance at the client side is mandatory, to minimize the number of jobs/applications which have to be restarted after a transient or partial server side problem or any kind of network glitch or damaged files

[4]Systems that consist of a combination of storage hardware and storage management software

Chapter 3

XROOTD

All of these requirements listed above complies the xrootd system (”eXtended rootd”).
Its structure 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 namespace.
These kind of structures in any case present an interface defined as a communication protocol, which defines the possible interactions and the functionalities given to the clients. The specific client of the xrootd system has been built in ROOT (XTNetFile) compliant form and and also as a POSIX compliant one.

Some of the design choices which give the needed functionalities are:

- **the communication protocol**, which defines an interface able to:
  - ◇ request access to an xrootd system through authentication handshakes (password based authentication, GSI authentication or Kerberos IV, V)
  - ◇ query a system for resource location
  - ◇ get access to the requested resource in the place where it can be accessed (i.e. servers giving access to local data or xrootd proxies allowing remote sites interoperability)
- **sophisticated communication policies** at the client side, able to handle any kind of communication errors (fig. 3.1)
  - ◇ The failing requests are retried until:
    - ⋆ another working server is found
    - ⋆ the same server becomes available again
    - ⋆ a specified maximum number of retries is reached
  - ◇ A read/write error is treated as a redirection
    - ⋆ To the first encountered load balancer (if any)

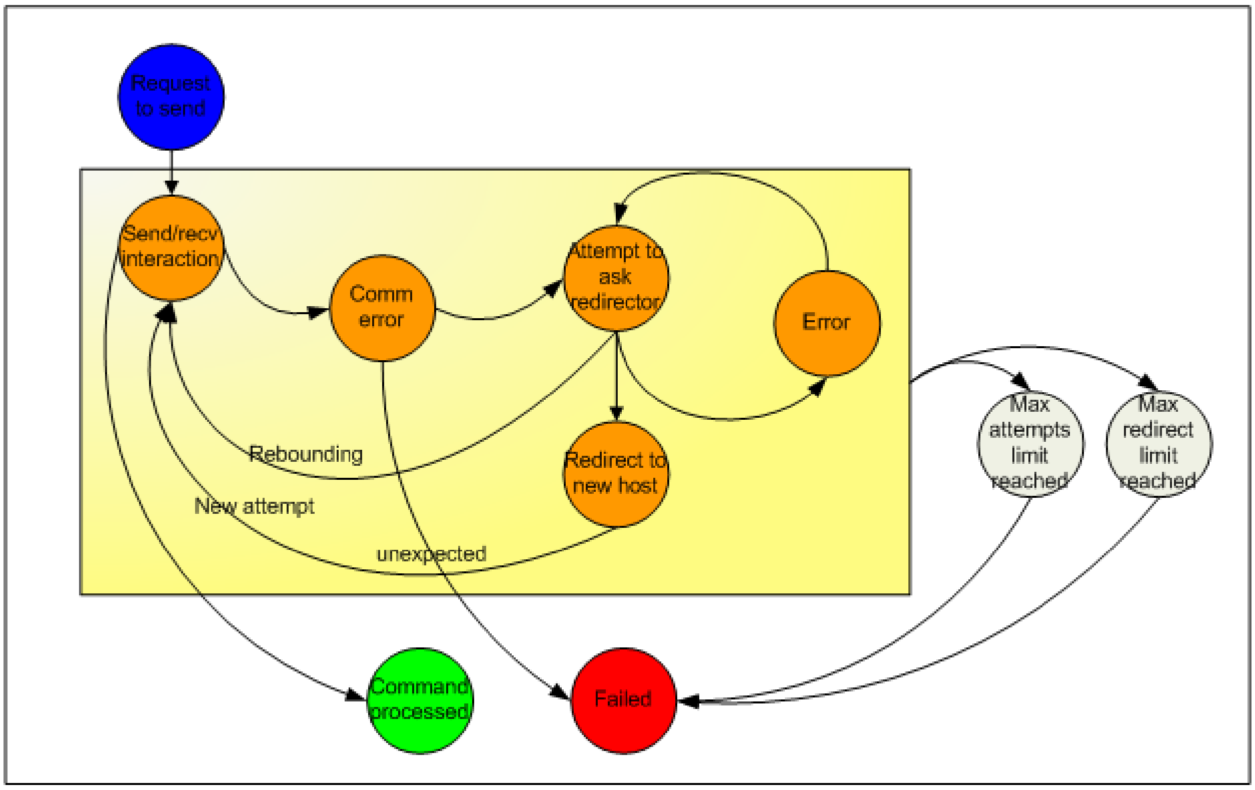

Figure 3.1: State transition diagram for the fault tolerant behavior of a client

  - ⋆ To the same server (rebouncing) if no load balancer
  - ⋆ Each reconnection attempt is counted as a redirection

- **Multiplexed persistent connections**: this means that a single TCP connection from a client to a server can carry multiple independent data streams for other client instances. Also, TCP connections are persistent for a short period if connected to a data server, for a long time if connected to a load balancer(redirector). This helps in lowering the system resource consumption and the network overheads due to repeated multiple connections to the same host.

3.1 XROOTD as layered architecture

The server side is composed of four layers:

- Network and thread management layer
- Protocol layer
- File system layer
- Storage layer

A layered approach allows optimizing a specific set of functionalities to minimize resource usage.
Additionally, each layer is sufficiently isolated so that it can be dynamically loaded. This allows for a number of implementations to determine which functions best in any particular environment. By extension, multiple implementations of a particular layer can be loaded at one time. This proved to be essential in providing backward compatibility with rootd since the thread management layer could run the rootd as well as the xrootd protocol at the same time[1].

3.1.1 Network and Thread Management

This layer is essentially the protocol dispatcher. It isolates all other layers from the details of socket and thread management.

The particular optimizations that were incorporated in this layer were:

- **Socket stickiness**: A socket is temporarily bound to a thread and associated objects to avoid rescheduling overhead between requests. The socket is unbound when it becomes inactive or when system resources become too constrained to allow such dedication of resources.
- **Socket polling**: The mechanism used to detect sockets ready for I/O is made architecture dependent. For instance, /dev/poll, a more efficient polling mechanism, is used on those platforms that support it. Otherwise, poll() is used but is optimized to significantly reduce the processing cost of the poll table.
- **Object persistence**: Allocated objects remain allocated as long as there is a reasonable potential for reuse. This includes thread objects. Managing objects in this way optimizes memory usage while minimizing synchronization points where objects are allocated and deleted.
- **Generalized object scheduling**: the major object at this layer is "job" class. Most object derive from the job class and objects at other layers generally do so as well. Any job class derived object can be asynchronously scheduled to perform internal maintenance and tuning functions and is extensively used for global optimization purposes.

3.1.2 Protocol Layer

The xrootd protocol is the default protocol run by network and thread management layer. This 64-bit TCP-based protocol provides generalized file access that in many ways is similar to AFS.

However, the protocol has been optimized in several ways to be more scalable:

- Multiple independent streams are supported on a single socket. This minimizes resource usage in the presence of multiple requests.

[1]the selection is determined by the level of client software being run

- Clients can be redirected to another server at any time. This allows dynamic server selection and load distribution while providing for direct point-to-point client-server connections.
- Clients may be asked to delay server contact. This allows the server to coordinate overloads and resource constraints by pushing the problem back into the network; avoiding typical server meltdowns when faced with a client onslaught.
- Clients may piggyback read-ahead lists with any read request. This allows the server to optimize future disk access and provide better performance.
- Client may ask for files to be prepared for future access. This allows the server to make sure files are online and properly placed for future client requests.
- Servers may ask clients to perform certain actions at any time. This is known as unsolicited response requests that typically take the form of redirects and execution deferrals; providing the server maximum flexibility in coordinating the load.

Additionally, the protocol provides for a generalized security framework that allows any authentication protocol to be used; providing scalable security.
The scalable protocol elements also provide the foundation for the inclusion of peer-to-peer capabilities. Here, contacts would contact a distinguished server that would search for the best possible source of the requested data and then direct the client to the corresponding server. Should that server become unavailable, the client is always free to launch another search.
Because traditional client-server interactions and peer-to-peer model are incorporated in the same file access protocol, it is possible to cover the full range of file access architectures; maximizing the potential for scalability.

3.1.3 File System Layer

The file system layer provides an implementation independent view of a file system. It is at this layer where peer-to-peer file-access decisions, if enabled, are done. This allows uniform access in a variety of situations.
Additionally,

- Multiple requests for the same file are merged.
- Redundant file operations across multiple clients are screened out.
- Idle files are closed after a timeout.

This layer also provides file-based access control based on the authentication information provided by the protocol layer.

3.1.4 Storage Layer

The storage layer provides the particular implementation of a logical file system. Here, logical file system operations are translated to specific actions optimized to underlying storage.

Some of the enhancements provided by this layer are:

- **Transparent access to multiple file systems** Here, any number of file systems can be aggregated to provide for a single view of storage. File system aggregation allows one to increase the number of actuators and provide greater flexibility in the physical placement of files for increased performance.
- **Mass Storage System integration** Here, near-line or off-line storage can be added to enhance the capacity of on-line space. Files are transparently staged and de-staged from on-line disk.

3.2 Individual components of XROOTD

The xrootd server architecture is shown in fig. 3.2. It is composed of multiple components. Each component serves a discreet task and is easily replaceable.

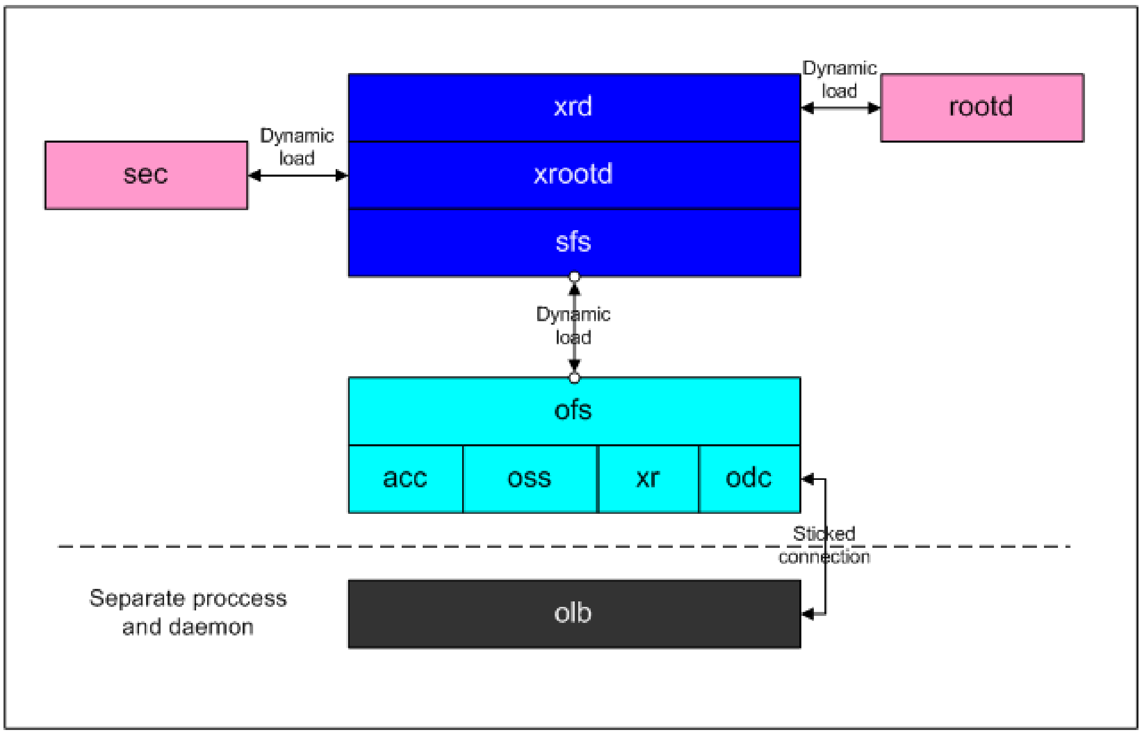

Figure 3.2: XROOTD components architecture

3.2.1 The xrd Component

The xrd component provides networking support, thread management, and protocol scheduling. This component has the potential to severely impact server scalability. Careful attention was given to algorithms used to insure minimum overhead per client-server interaction.

The performance oriented features include:

- Use of the best socket handling features that the underlying OS provides.
- Threads are managed by a lightweight scheduler.The scheduler attempts to keep enough threads ready to handle the recently experienced load. Generally, threads are created for each incoming request, up to the maximum computed based on system resources. The exact number can also be configured. Once the maximum is reached, threads are shared by all clients. When threads become idle, they are automatically eliminated using an exponential decay function.
- The xrd component allows multiple protocols to be used at the same time. Each configured protocol is asked whether it can handle an incoming connection. A protocol object instance is created once a match is found. This object is then scheduled, when necessary, to handle client interactions. This feature is used to provide simultaneous xrootd and rootd protocol support.
- The xrd component provides for the highest level of parallelism by avoiding functions that tend to serialize execution, maintaining suitably grained locks, and using threads whenever possible to perform internal housekeeping. As such, it implements a very low overhead protocol engine capable of serving thousands of clients.

3.2.2 The xrootd Component

The xrootd component implements the xrootd protocol.

This protocol provides generalized POSIX-like file access enhanced by High Performance Computing (HPC)[2] extensions, and fault recoverability (FR) features.

The protocol is architected as a platform-neutral simple binary stream to eliminate most of the encoding-decoding overhead associated with other similar protocols. While this reduces the chances of general inter-operability, the HPC and FR extensions make the protocol unlikely to inter-operate with other network based file protocols without sacrificing significant usability.

The significant HPC extensions include:

- **asynchronous responses** so that a client can launch multiple requests at the same time, asynchronous I/O when the operating system supports it and resources are available
- **pre-reading data** so that it is available in memory on the next client interaction
- **asynchronous file access preparation** to minimize file open overhead

[2]the field of high performance computing (HPC) comprises computing applications on (parallel) supercomputers and computer clusters

- **automatic I/O segmenting** to allow data stream multiplexing
- **client-directed request monitoring** to allow application tuning

As can be seen, most of the HPC features involve enabling a rich set of asynchronous facilities to provide clients the maximum number of opportunities for parallelism.

The significant FR features include:

- request redirection so clients can be dynamically steered to least loaded operating servers
- request deferral so that the server can even out highly variable loads without increasing its own state overhead
- client-directed error state notification
- unsolicited responses to asynchronously reconfigure client-server connections

The FR features are used to implement dynamic load balancing and server failure recovery. Most of these features are used in combination with the Open Load Balancing (olb) structured peer-to-peer (SP2) system that can be run in conjunction with xrootd. The olb system is described below of this report (see section 3.3).

In addition to providing xrootd protocol file access, the xrootd component is also responsible for invoking the authentication component.
The authentication component is structured as a highly versatile multi-protocol suite.
It allows to dynamically load following security protocols:

- host-based authentication
- GSI authentication
- Kerberos IV authentication
- Kerbors V authentication
- password-based authentication

3.2.3 The sfs Component

Since file serving is the focus for xrootd, it interacts with another component to provide file access. This service is based on the Standard File System class, XrdSfs. The actual implementation of this class is loaded at run-time; allowing for numerous implementations, as needed by any particular installation.
A default implementation is provided that provides the minimum set of features to support the xrootd protocol. Another configurable implementation is also provided. This implementation supports all the xrootd protocol features and is called the Open File System (ofs) component.

3.2.4 The ofs Component

The ofs component provides enhanced first level access to file data. Since this component is expected to support the full set of xrootd protocol features, it is architected as a multi-component service. Each component is responsible for implementing a particular set of features that can be easily re-implemented to correspond to the actual underlying architecture.

These components are:

- Access Control (acc based on the XrdAcc class)
- Open Distributed Cache (odc based on the XrdOdc class)
- Open Storage Service (oss based on the XrdOss class)
- Peer proxy service (xr based on the XrdXr and XrdOss classes)

The ofs component is responsible for coordinating the activities of these components to provide an effective file system view.

3.2.5 The acc Component

The acc component implements the authorization service. This service is responsible for granting clients access to files. It uses the authentication information, if any, passed through by the xrootd component.

The authorization component is implemented as a reverse file capability list. A capability oriented implementation was chosen to optimize operations when the number of files substantially exceeds the number of users capable of accessing files. In this scheme, each user and user association can be granted or denied access to files that start with a particular prefix. The set of privileges correspond to those implemented by Windows XP and is a super-set of POSIX privileges.

Generally, this provides the ability to associate capabilities (or lack of capabilities) to users. It is a reverse file capability list in that specifying a file prefix completely effectively implements an access control privilege scheme where a file (or set of files, directory) is associated with a number of users and their capabilities.

Authorization information may come from various sources, depending on the installation, not only from a file. Regardless of the information source, the following semantics prevail.

1. An authorization database file contains one or more authorization records.
2. Each authorization record is considered to be a capability.
3. Each capability is tied to a unique identifier within its name class.
4. An identifier may be a

- host name
- domain name
- netgroup name
- template name [3]
- unix group name
- user name

5. All template identifiers are logically processed in the order specified. The processing order of other identifiers is immaterial.
6. Each identifier is associated with an arbitrary list of path prefix-privilege pairs as delete, insert, lookup file etc.
7. The list is always searched from left to right (i.e., in the order that it was specified).
8. The privileges associated with first prefix that matches an incoming path name are considered to be the applicable privileges, next is ignored

3.2.6 The odc Component

The odc component is responsible for locating the right server to use for a particular file open request. It is invoked by the ofs component when dynamic load balancing or proxy support is configured.
The odc provides numerous services under the guise of finding the right server for the requested file.

The four main functions are:

1. communicating with the olb to discover the location of a file and appropriate server to provide access to that file
2. passing xrootd protocol requests to the olb that may need to be handled on a remote host (e.g., file preparation, file removal, etc)
3. coordinating the activities of other xrootd servers running on the same host
4. initiating the use of a proxy service should remote file access be needed

When invoked, the odc may respond with a server-port pair indicating that the client should be redirected to that host for subsequent file access. This occurs when the server is configured in **"redirect remote"** mode.

The odc may respond with a simple port number, indicating that the client should be redirected to another xrootd server running on the same host. This occurs when the server is configured in **"redirect local"** mode.

[3]it allows to define template for group of users and use it in the other authorization record

The odc may respond with an instance of a oss object that should be used for actual file access. This occurs when the server is configured in **"redirect proxy"** mode.

When the server is configured in **"redirect target"** mode, the odc passes execution state information to the local olb. That information is used in redirection decisions by other olbds serving xrootd configured in **"redirect remote"** mode.

Finally, the odc may respond with a null response indicating that the incoming request should be processed by the oss component as if the odc was not configured.

For redundancy, the odc can communicate with multiple olbd's in order to provide a fault tolerant environment as well as to load balance requests among all of the olbd's.

3.2.7 The oss Component

The oss component is responsible for providing access to the underlying file system. It is invoked by the ofs component to perform actual I/O as well as execute file system mete-data operations (e.g., rename, remove, etc).

The difference between the phrase file system and storage system is that a storage system provides access to stored data that may or may not be reside in an actual file system. For instance, the file may reside in a Mass Storage System (MSS) and will need to be retrieved prior to the access. Alternatively, the file may reside on another xrootd server and a proxy relationship will need to be established in order to provide access. In all cases, the actual act of providing access to data is handled by the oss.

The mechanism used to provide that access is encapsulated by the oss to provide a uniform view of storage regardless of how it must be accessed:

- access to an MSS using configurable agnostic callouts so that any kind of MSS can be used
- a generic file system cache facility so that multiple file systems can be aggregated into a single uniform view
- a proxy service to provide real-time access to files that reside on other xroot servers
- I/O and meta-data access to a UFS-type file system
- asynchronous I/O capabilities, should then operating system support it

3.3 Load balancing, additional performance and fault tolerance

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 the 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 interdependent. That is, while servers can be aggregated, a failure of any server should not affect the functioning of other servers that participate in the scheme.

As the best is shown the model of peer-to-peer systems which have shown to be extremely tolerant of failures and scale well to thousands of participating nodes. The structure consists of one or more servers, called redirector and supervisors, rooting a B64 tree structure of data servers (fig. 3.3).

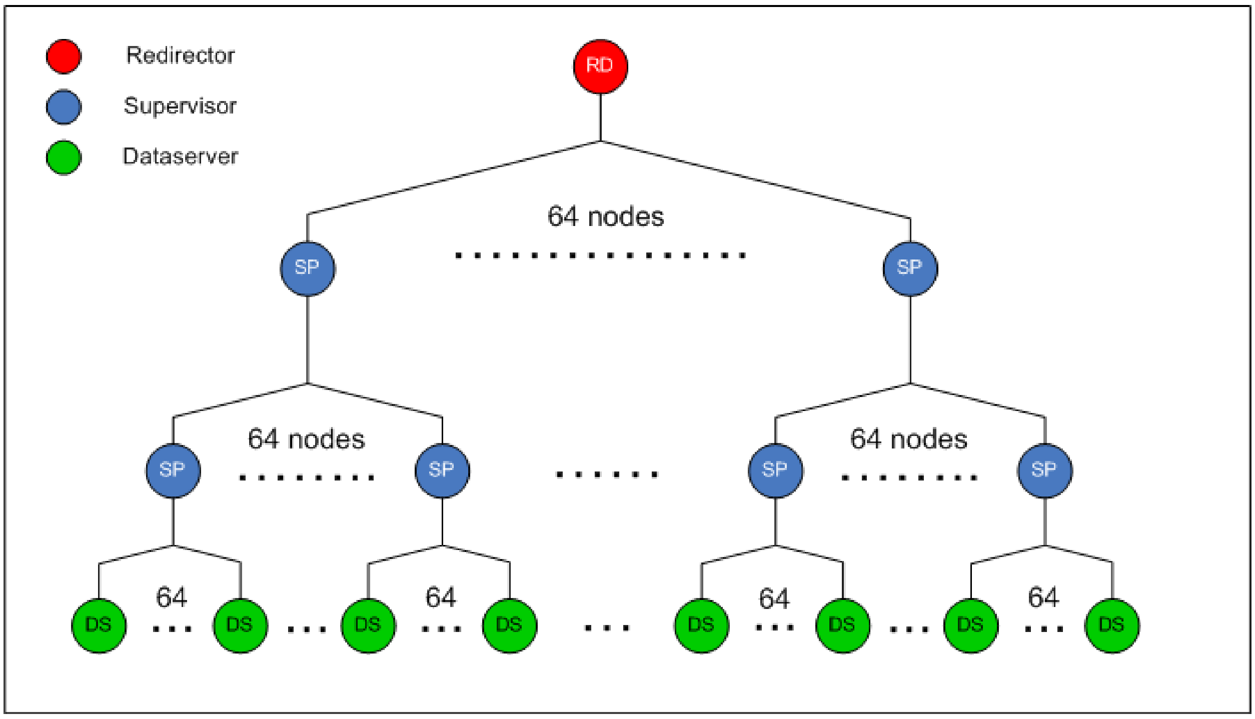

Figure 3.3: The B64 structure of xrootd servers

Information flows up the tree to the redirector or supervisors that redirect client requests to servers lower in the hierarchy. From the server's perspective, data servers only know that they are participating in a cooperative structure but no single data server is aware of the structure. Servers at the root of each B64 node only know the existence of their immediate neighbors and one or more servers higher in the tree. This effectively isolates failures to small areas within the configuration. Even the most significant failure in the structure only causes a small part of the overall structure to reconfigure in order to maintain a cooperative data access view.

A hierarchical model was chosen because this minimizes the number of messages that needed to flow through the system and creates predictable access paths. The choice of a B64 tree was done out of practical necessity to keep the decision making overhead to reasonably low levels; avoiding latency pile-ups that could cause the system to become unstable. This is called as a structured peer-to-peer model because while servers work in a peer-to-peer fashion within the system, a particular structure is imposed.

The system provides high levels of scalable performance because clients can be dispersed throughout a large set of servers. Adding additional servers naturally allows more clients to participate. This happens because clients will either be directed to servers that have the requested data or should those servers near saturation levels, clients are directed to less active servers that will replicate the requested data. Hence, the load will be balanced across all of the servers. Unanticipated hot spots are naturally alleviated because the protocol allows any server that finds itself in a hot-spot to redirect clients away from itself. This forces clients to settle upon other servers that are less loaded. This is called as a mode of operation ”dynamic load balancing” and it is one of the major reasons that the system scales.

Since the protocol allows connection configuration changes to occur at any time, the system also provides unprecedented fault tolerance. Should a server failure occur, a client needs only to contact a redirector to find an alternative source of the data.

3.3.1 The olbd Process

As I mentioned in the previous section, the system was designed using an independent set of servers to provide the control information to effect xrootd server selection. This set of independent servers forms what we can call the **control network**.

It is logically independent of the data access services provided by the xrootd servers; which form the **data network**.

The control network is made of servers called Open Load Balancing Daemons (olbd), Each node that provides data must have an xrootd server and an olbd running on it. The olbd runs in **server** mode since it is responsible for relaying information about the node providing a data service.

The xrootd server running on a data node connects to the local olbd. This allows the olbd to know the status of the server and the port number that it is using. This information is relayed to other olbd’s so that clients can be properly directed to the data node, as needed.
Each local olbd subscribes to one or more olbd’s running in **manager mode**. A subscription effectively tells the target olbd that the node is capable of providing a data service on a particular port. Identification of manager olbd’s is done by administrative configuration. In case of fault tolerance and server’s crash in manager node is prepared a logic of round-robin.

A manager olbd is special in that it resides at the topmost level of the connection hierarchy. It differs from server mode olbd’s in that it accepts connections from multiple xrootd servers. According to the fig. 3.3, clients making requests of these servers are always redirected to appropriate servers lower in the hierarchy. Redirector does not provide data only request steering information, but supervisors can hold a data as such can steer clients. Supervisors are entering the scene in case of more than 64 data servers cluster and help to the redirector with managing the whole cluster.

When a manager olbd is asked for request guidance, it first checks its cache of recent requests to see if it already knows where the request should be sent. If the information exists in its cache, the response is immediate. Otherwise, the redirecting xrootd server[4] is told to delay the client for a fixed period of time while it asks the olbd's that are subscribed to it whether or not they have the requested file. All olbd's that have the file report its existence. Those olbd's that cannot find the file on their node stay silent. File existence information is collected by the manager olbd and cached. Eventually, the client comes back and asks for the file which prompts xrootd server to ask again. This time the information is in the cache and the response is immediate.

3.3.2 Example of interaction in the xrootd structured peer-to peer system

The following figure 3.4 illustrates a simple minimal system of interacting xrootd and olbd process during the client request. In the diagram, there are three hosts: **x,y** and **z**.

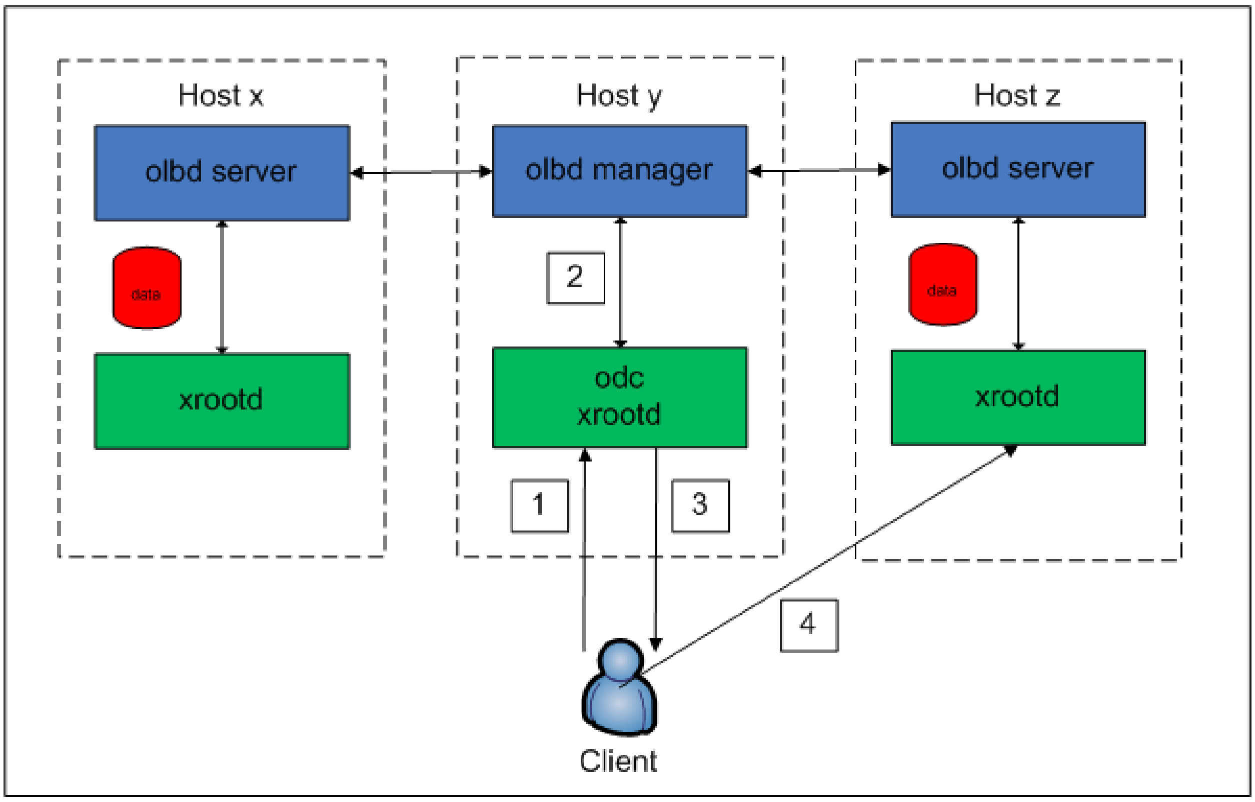

Figure 3.4: Client-server dynamic load balancing

[4]xrootd on the node where olbd runs in manager mode

Host **y** serves as the load balancer. Host **x** and **z** the are the hosts that can be used to serve data to clients and load is distributed between the two. Consequently, host y runs in manager mode and hosts x and z run in server mode. The servers are connected to the manager and provide load information. The xrootd running on host y connects to the manager as well, using the **odc** component. However the xrootd on host y uses the manager to determine which server to direct client request.

The typical open request is handled in four steps:

1. The client directs the open request to the **xrootd** that runs on the **manager's** host.
2. The **xrootd odc** component asks the **olbd** manager which machine is the best machine using a variety of configurable parameters.
3. The xrootd on host **y** tells the client, in this example, that host **z** is the best host to use for the file.
4. The client then redirects the request to the **xrootd** running on host **z**.

During the subscription process of data servers to manager host, each server indicates the file paths to which it is willing to provide data access. Periodically, the manager **olbd** requests load information from each server. Each server reports CPU, network I/O, memory, paging load as well as free space. The information is used to select the best server for an open request.

3.4 The xrootd's benchmarks

In order to test the real-world performance of the server, a series of analysis jobs were run against a single file, allowing data to be served from the file system memory cache and avoided disk-speed anomalies that make performance results hard to interpret. The CPU-intensive work in the analysis job was removed to force the maximum possible request rate from each client while preserving the original data access pattern. Thus an event in this context represents a bounded series of server transactions.

The red line in fig. 3.5 shows that the number of events per second scales linearly with the number of clients.
The graph also shows that the rate of increase unexpectedly slows after about 200 clients. This effect is a benchmark induced aberration. The first two hundred clients were each run on a dedicated machine; after which up to two clients were run on each machine. Running more than one client on a machine adversely affected each client machines performance by 9.7%. This loss of efficiency appears as a deviation of the expected event rate.

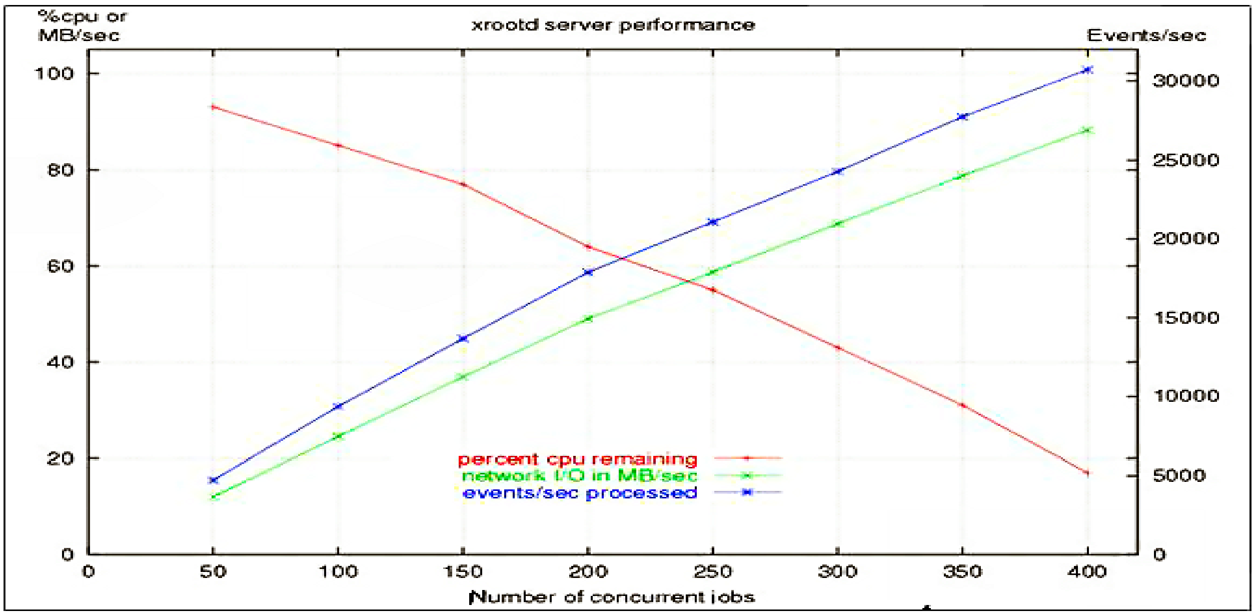

Figure 3.5: Throughput and resource consumption versus number of clients for a single server

3.4.1 NFS vs xrootd comparison

During the performance testing was also measured network I/0 and time to open the file according to the comparison of xrootd and NFS.
In the figure 3.6 is visible that with increased number of jobs is the network I/O constant. The time NFS to access the file is linearly increased compares to the time in xrootd which is almost constant in case of smaller measure.

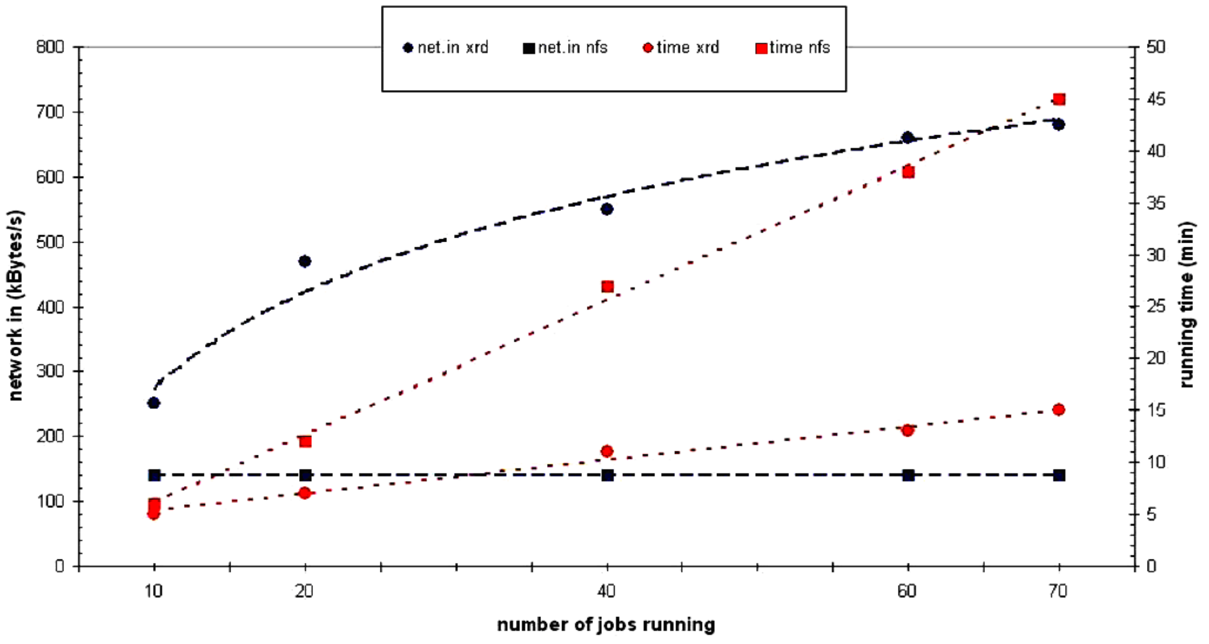

Figure 3.6: NFS vs xrootd comparison

Chapter 4

Conclusion

At the beginning of my report I presented the goal of distributed computing, why this section of the computer research is necessary and important for the rest of research branches and next evolution of technology.
In the next section I explained by what are the physicists from the STAR experiment interested in, why they need data oriented distributed computing and also described tools and technologies which are used for particle analysis. I summarize all requirements on data distributed system, as scalable, high performance, fault tolerance etc.
The bigger part is dedicated to one of the data distributed system, called XROOTD. I introduced all features of XROOTD, listed the whole architecture from the point of layers and then from the component's angle of view. This report is ended with some benchmark's results of this system.

Bibliography

[1] BRUN, F.; RADEMARKERS, F.: ROOT documentation, 1990.

[2] STERN, H.; Managing NFS and NIS: O'Reilly & Associates, 1992.

[3] DORIGO, A.;ELMER, P.;FURANO F., HANUSHEVSKY, A.: XROOTD - A highly scalable architecture for data access, SLAC, 2005

[4] HANUSHEVSKY, A.;STOCKINGER, H.: Proxy Service for the xrootd Data Server, SLAC, 2004

[5] HANUSHEVSKY, A.: XRD Configuration Reference, SLAC, 2005

[6] HANUSHEVSKY, A.: Open File System & Open Storage System Configuration Reference, SLAC, 2005

[7] HANUSHEVSKY, A.: Open Load Balancing Configuration Reference, SLAC, 2005

[8] HANUSHEVSKY, A.; GANIS, G.: Authentication & Access Control Configuration Reference, SLAC, 2005

[9] HANUSHEVSKY, A.: Cache File System Support MPS Reference, SLAC, 2004

[10] MINGLU, L.: Grid and cooperative computing : Second International Workshop, GCC 2003, Shanhai, China, 2003

[11] http://www.hpss-collaboration.org/

[12] http://www.star.bnl.gov/STAR/comp/