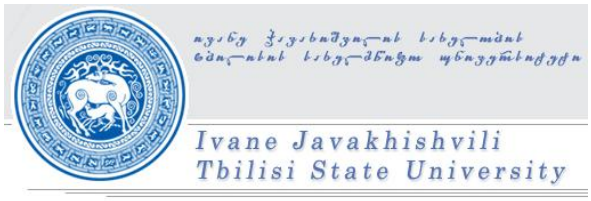


Federated Metadata Search in UNICORE

Master's Thesis of

Konstantine Muradov

Supervisors: Prof. Dr. Manana Khachidze (TSU)

Dr. Bernd Schuller (FZJ)

Faculty of Exact and Natural Sciences
Module of Information Technologies

Master's work carried out at the
Jülich Supercomputing Centre (JSC) of
Institute for Advanced Simulations (IAS), Forschungszentrum Jülich GmbH, Germany

2013

The computer revolution is a revolution in the way we think and in the way we express what we think. The essence of this change is the emergence of what might best be called procedural epistemology--the study of the structure of knowledge from an imperative point of view, as opposed to the more declarative point of view taken by classical mathematical subjects.

- H. Abelson and G. Sussman (in "The Structure and Interpretation of Computer Programs)

Table of Contents

Acknowledgements

This work would have not been possible without the support and encouragement of many people, both computer scientists and non-computer scientists domains. Hence, I would like to take this opportunity to express my sincere gratitude to all of them who have always been with me and for me.

I want to thank Prof. Doc. Hans Ströher and Doc. Andro Kacharava for all their support starting from Georgian-German workshop in basic science held in Tbilisi in 2012. For the invitation to visit Forschungzentrum Jülich in 2012 helped me a lot to become familiar with research held at Jülich Supercomputer Center (JSC), and help to find contact in JSC. Their continuous support throughout the duration of my stay and rich advices helped me a lot.

I wish to sincerely thank my supervisor, Manana Khachidze, for introducing me to computer science. Her introductory lectures gave me insight into processes, I was analyzing. I'm sincerely grateful to her for teaching me how to doubt acquired knowledge and results and how to come back to what is thought to be understood and how to rethink it one more time. She has always given me an exemplary guidance to develop the necessary skills and the way to cultivate the ability to think independently as a graduate student.

I want to thank Bernd Thomas Schuller for his agreement to be my supervisor for his help and willingness to answer my questions both during my visit in Juelich and prior to that in Tbilisi, which was always in touch. He showed me software engineering in scientific angle, for introducing in grid distributed computing.

I want to thank my close friend Vazha Ezugbaia who helped me to become familiar with grid computing and other technological issues to write this thesis. I also want to thank the people with whom I had become friends and who has always given very precise and valuable advices in thesis writing - Phd students of IKP (Institut für kernphysik – Institute of nuclear physics) Malkhaz Jabua and Zara Bagdasaryan, they also helped me to adapt in Germany.

Finally, and most importantly, I would like to thank my parents, whose constant love and support cannot be matched with any word. They taught me that knowledge is the best treasure, you can gather during your whole life, and they showed me by their own example how it is to do everything with great love and passion. I'm infinitely grateful to them for having the greatest effect on the formation of my personality, always giving me the greatest lessons and great encouragement, and at the same time letting me take responsibility for my decisions and grow into an independent person that I am now. I admire and love them dearly, and dedicating this thesis to them is the smallest thing I can do.

I also want to thank my sister (G.Muradova), as well as my friend (V.Engibaryan) for being very supportive during all this time and proud of me, encouraging me to achieve more. I would like to justify the hopes pinned on me by my family and teachers, to make my small contribution to the development of science and to be a contributing member of my country, and to society in general. I think my thesis allowed me to do just that.

To my wonderful parents,

R.Muradov and G.Khatoeva.

Abstract

The purpose of this document is to familiarize readers with the work, which was done in the frame of master's thesis "Federated meta data search in UNICORE" at the research center “Forschungszentrum Jülich ” at the Juelich Supercomputer Center - “JSC” in the terms of memorandum between Rustaveli National Science Foundation and Jülich Research Centre to support cooperation among both research institutions (including Tbilisi State University) and their scientists.

This paper introduces “Federated metadata search” solution for managing large amount of data produced in a Grid. In order to manage the increasing amount of data produced in Grid environment, a highly scalable and distributed Grid storage system is needed. Howerer, the simple storage of data is not enough. To allow a comfortable browsing and retrieving of the data, it is crucial that files are indexed and augmented with metadata.

In this document it will be described step by step process of solving problem of federated meta data search in UNICORE grid platform. During this master thesis this topics were addressed and the following tasks were performed:

Specification of use cases and functional requirements

Design of the federated search service

Web service interface

Query specification (search options, limiting to certain sites etc...)

Result collection and Result set management

Implementation of the federated search service as a UNICORE WSRF web service

Setup of working environment.

Implementation including unit tests

Validating and functional testing

The result of this work is developed feature for UNICORE which gives user comfortable tool for distributed data search, generated by platform during file processing.

The task itself would be described further would be presented several possible solutions to this problem – the pros and cons of each. Hereinafter

it will be described in details chosen solution and the reasons for selecting option, the software development process, features added that were not part of the original problem, but have been added to provide comfort to the user, the process of testing and bug fixes, results and statistics.

At the beginning there will be brief introduction about Grid computing , UNICORE and meta data support in UNICORE and what was the need of formulation of the problem of this thesis and after the process of implementation the task. .

Introduction 1

The experience with the grid deployment shows that storage is similarly important as compute resources. Moreover, recent developments in the cloud technology allow users to access distributed storage in easy and efficient way. Users get access to the different cloud storages such as Google Drive, Dropbox, or Globus Online.. Such solutions differs in technology but for the user they are visible as folders on the desktop or as a simple web application allowing for easy (often drag and drop) upload and download files.

Easy configuration and lack of the access barrier makes such solutions very popular for non-experienced users. Utilization of the cloud technology in the everyday applications and devices such as editors or tablets provides cloud storage additional popularity despite limited security or lack of control.

Unfortunately, grid middleware is focused on the CPU utilization and still lack storage solutions which offer functionality similar to the cloud storage. The access to the data is organized according to the computational requirements rather than users' need.

For today it is difficult to find software which does not implement process or store data. With increasing amounts of (distributed) data made accessible to the users, the process of locating particular pieces of content becomes harder and harder. This is amplified by the fact that the “data production” is often a joint effort of many parties spread all over the world. So that the resulting data are frequently stored in various file formats or file hierarchies and no common naming convention can be taken for granted. Locating a particular content under these circumstances is like looking for a needle in a haystack or, considering the far-reaching distribution, in multiple haystacks.

Since the amount and size of files in a single storage can be arbitrary high (or it can be not text – other type of data), it is not feasible to perform a full-text search in all files. A much better way of getting to grips with the data is to provide a way for describing data resources. Such “data on data” or metadata can describe the content of particular files independently from their representation (file format or naming convention used), physical locations, or origins. Since the metadata will be smaller than the content it describes, it can be indexed and searched much more efficiently.

Data in themselves are too important but also it is important their correct management. A huge amount of data without correct management always will be useless – in petabytes of data if there is no search engine, then as you probably guessed, these data becomes difficult managed and often useless. Search data - which includes the pre-indexing for fast results – is not a trivial task, but it can be more complicated if they are stored in different media and consequently there is no centralized data index, which, in case of need, can be used to find out in where data is located.

Grid technologies are starting to realize their large potential to provide innovative infrastructures for complex scientific and industrial applications. The UNICORE Grid computing solution provides a seamless, secure, and intuitive access to distributed Grid resources. However, to make Grids more useful for knowledge-oriented applications, more effort in the fields of semantics, metadata and knowledge management in distributed, heterogeneous environments is needed.

UNICORE is grid – distributed platform and consequently it does not have centralized data storage. Each node stores its data in local media, it can be data of one user which were stored during

several calculations. In UNICORE each node has its own index file of stored data, which is provided by ”Lucene Indexer”, but there is no possibility of distributed search, in case of need user have to provide search for each node separately - which is inconvenient for two reasons:

- User has to do same operations for all nodes.
- User can't get summary result of search - he gets separately results from each node.

The goal is clear – to provide federated search and return union result with some useful statistical data. Further in this document it will be described process of design and implementation of this task.

For user convenience, it was decided that he/she should be able to define search criterias such as:

- On which storages should be provided search.
- Search query – key words.
- Date of data creation – upper and lower limits

The process of designing of this structure and implementation would be described in section XX.XX. User should be able to watch process of data searching - at any time find out status and progress of search.

Figure 1 User data and metadata management [2]

Also it was decided, except the search results themselves – storage location and resource address, it should be returned statistics about search such as:

- Search time.
- Processed servers count.
- Amount of results.

After goal design and structure is defined begins next step – task implementation. To realize task in UNICORE it was necessary to take in consideration several rules/steps of adding modules/futures. For this task it was necessary to take into account next rules/steps:

- Implement task in Java using JDK 6.0
- Define web interface.
- Implement UNICORE WSRF (using JAX-WS Document style)
- Unit tests (using JUnit)
- Validating and functional testing

The above described steps are necessary because UNCORE has service based N-tier architecture - where each module exchange information using web services.

In this document also will be considered and illustrated other possible - declined variants of designing current task and the reason of their rejection.

Before describing process of task implementation, it would be presented brief introduction of Grid computing - because UNICORE is a grid platform and it is the reason why data is distributed between several storage, after it would be short introduction to UNICORE – architecture, how it is implemented and how it is organized, to illustrate for readers why some implementations were made one way and not another.

Grid computing 2

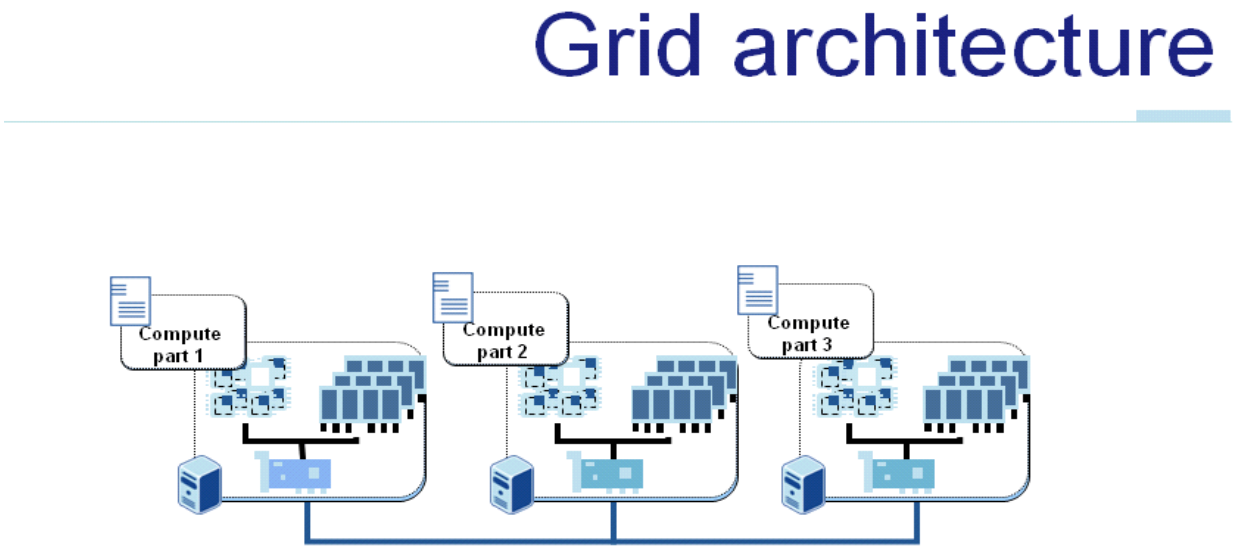


Figure 2 Grid infrastructure [3]

The term grid is referring to a distributed computing infrastructure, able to provide resources based on the needs of each client. Grid technology can largely enhance productivity and efficiency of virtual organizations, which must face the challenges by optimizing processes and resources and by sharing their networking and collaboration. Grid computing technology is a set of techniques and methods applied for the coordinated use of multiple servers. These servers are specialized and works as a single, logic integrated system.

The grid has developed as a computing technology that connects machines and resources geographically dispersed in order to create s virtual supercomputer. A virtual system like this is perceived like it has all the computing resources, even if they are distributed and has a computing capacity to execute tasks that different machines cannot execute individually.

In the past few years grid computing is defined as a technology that allows strengthening, accessing and managing IT resources in a distributed computing environment. Being an advanced distributed technology, grid computing brings into a single system: servers, databases and applications, using specialized software. In terms of partnership between organizations, grid technology may include the same enterprise organizations as well as external organizations. Therefore, grids may involve both internal and external partners, as well as only internal ones [1].

The complexity of the environment in which the grid will be developed and the requirements that have to be applied depend on the environmental impact of trade, defining the relationship of trust, security considerations, globalization and integration period, of the company involved, in the market.

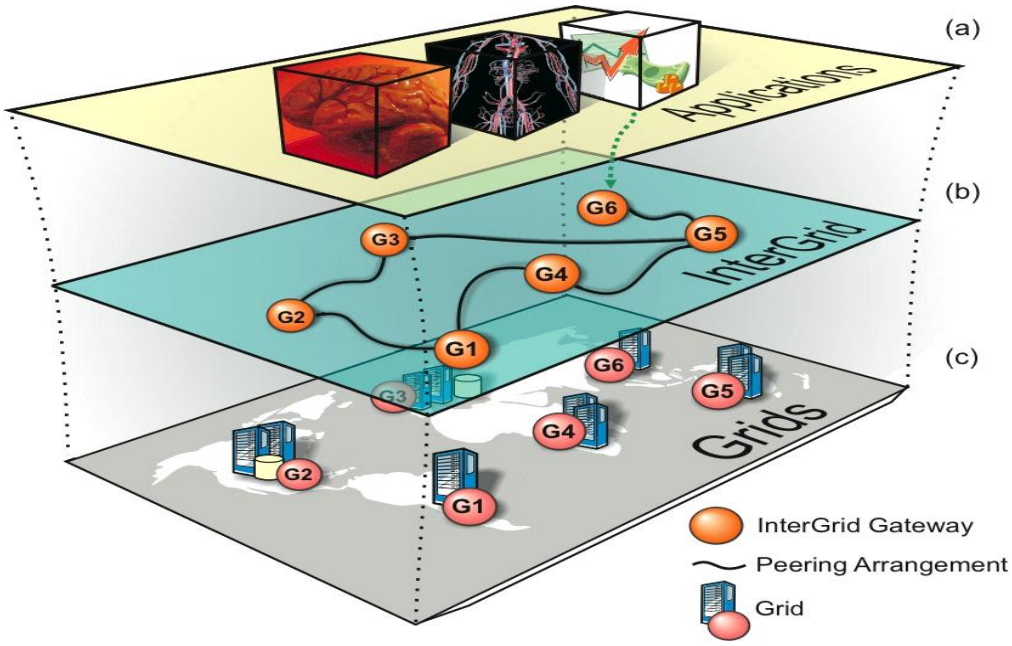


Figure 3 Inter grid [4]

Based on the different levels of complexity for the enterprise, grids can be categorized as follows:

1. Infra-Grid – this type of grid architecture allows optimizing the resource sharing within a division of the organization's departments. Infra-grid forms a tightly controlled environment with well defined business policies, integration and security.
2. Intra-Grid – it's a more complex implementation than the previous because it's focused on integrating various resources of several departments and divisions of an enterprise. These

types of grids require a complex security policies and sharing resources and data. However, because the resources are found in the same enterprise, the focus in on the technical implementation of the polices.

3. Extra-Grid – unlike intra-grid, this type of grid is referring to resource sharing to / from a foreign partner towards certain relationships is established. These grids extend over the administrative management of local resources of an enterprise and therefore mutual conventions on managing the access to resources are necessary.
4. Inter-Grid – this kind of grid computing technology enables sharing and storage resources and data using the Web and enabling the collaborations between various companies and organizations. The complexity of the grid comes from the special requirements of service levels, security and integration. This type of grid involves most of the mechanism found in the three previous types of grid.

Grid computing technology is not the only distributed technology; among other distributed technologies we can include: web technology, peer-to-peer, clustering and virtualization.

The similarities between the grid computing technology and the others distributed technologies are:

1. A grid hides its complexity, interconnects different resources and provides a unified perception of the environment.
2. Can share files and communicate directly through a central broker.
3. Clusters and grids groups the resources in order to solve a problem and can have a tool for unified management of all components. [2]

History 2.1

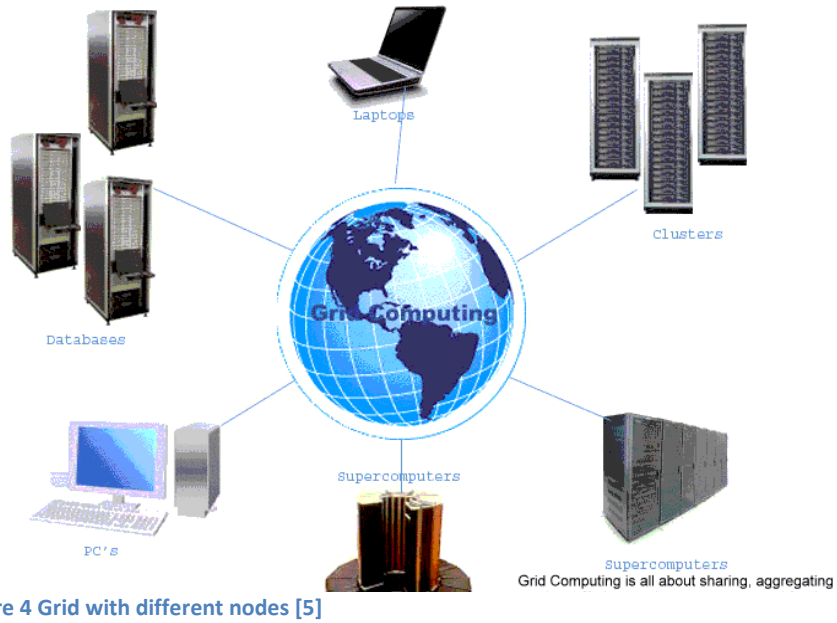


Figure 4 Grid with different nodes [5]

The term *grid computing* originated in the early 1990s as a metaphor for making computer power as easy to access as an electric power grid . The power grid metaphor for accessible computing quickly became canonical when Ian Foster and Carl Kesselman published their seminal work, "The Grid: Blueprint for a new computing infrastructure" (2004).

CPU scavenging and volunteer computing were popularized beginning in

1997 by distributed.net and later in 1999 by SETI@home to harness the power of networked PCs worldwide, in order to solve CPU-intensive research problems.

The ideas of the grid (including those from distributed computing, object-oriented programming, and Web services) were brought together by Ian Foster, Carl Kesselman, and Steve Tuecke, widely regarded as the "fathers of the grid". They led the effort to create the Globus Toolkit incorporating not just computation management but also storage management , security provisioning, data movement, monitoring, and a toolkit for developing additional services based on the same infrastructure, including agreement negotiation, notification mechanisms, trigger services, and information aggregation. While the Globus Toolkit remains the de facto standard for building grid solutions, a number of other tools have been built that answer some subset of services needed to create an enterprise or global grid.

In 2007 the term cloud computing came into popularity, which is conceptually similar to the canonical Foster definition of grid computing (in terms of computing resources being consumed as electricity is from the power grid). Indeed, grid computing is often (but not always) associated with the delivery of cloud computing systems as exemplified by the AppLogic system from [4]

Principles of development of a Grid System, applications and requirements 2.2

Grid systems must ensure the transparency of the following ways: access, location, heterogeneity, failure, replication, scalability, concurrency and behavior. Users and developers should not know exactly where an entity is located to interact with it and should not be forced to acknowledge the failure of

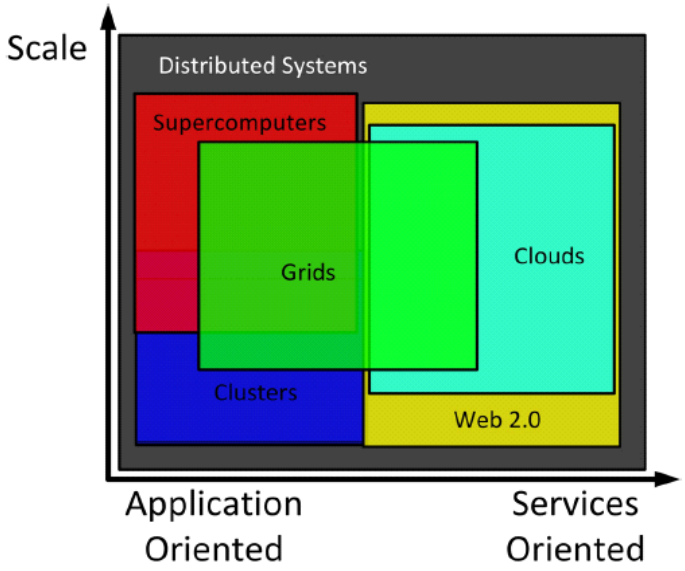


Figure 5 Distributed systems and grid [6]

components / resources of the grid system. This grid system must be equipped with self-recovery mechanisms in order to accomplish tasks. This represents the mask (transparency) on the vision system.

Grid architecture must be adaptable to as many solutions on the grid user needs and goals. A rigid system in which the known policies is limited, decision are pre-selected transaction cannot meet these demands. This grid system should allow greater flexibility for both semantic applications without imposing predetermined solutions to unique methods of implementation. Grid systems must also allow

users to choose the desired type and level of functionality and appeal to their own compromise between functionality, cost and performance. This philosophy should be integrated into the grid by specifying the architecture and functionality but not implementation of core aspects of the system. The core should therefore be made of extensible components, which can be updated and replaced whenever possible / necessary while implementing the system should provide an implicit development of each functional component useful to normal users.

In general one can speak in a grid environment for the existence of four categories of users: application users, application developers, system administrators and managers. Grid system architecture and implementation must be realized that it allows users to applications and their developers is focused on their task and not involve their direct work on issues related to installation and implementation of grid infrastructure, but can also be provided and ensuring access to it the grid infrastructure and related details if required.

Another aspect of flexibility to be ensured to grids is to keep the machine's operating system will be integrated into the system. Such grid systems must be developed so as to work with and above the machine's operating system and involve such changes, on configuration, but fewer in number and complexity.

Influence of resource management issues on application development:

1. Choice of appropriate resources. Choosing appropriate resources an application is made by the resource management mechanism that works in conjunction with a broker. This implies that the application to specify accurately the working environment required (OS, processor speed, memory, etc.). The application shows how dependent fewer specific platform the greater the chances that an appropriate and available resource to be found and resolved quickly workload.
2. Multiple computing sub-tasks. For applications involving multiple computing tasks to analyze the interdependencies between them should be made to avoid additional logic: the communication intercrosses, data sharing and tasks competing administration.
3. Managing computing tasks. If the application has to provide a reactive response to the user or to release resources when the application should be designed so as to use the mechanisms of grid resource management to ensure consistency and integrity of an environment.

The mechanisms for data management in a grid to maximize the best use of limited storage space, the network bandwidth and computing resources. The following aspects of data management must be considered in the development of grid applications:

1. Data set size. If the application works with data sets of very large ineffective, if not impossible, for data to be moved on the system that will run and load calculation. In this case one possible solution is data replication (copying a subset of the data set) on the system to execute computing tasks.

2. Geographical distribution of users, computing resources, data and storage. If the grid is geographically distributed environment and shows limited network speeds then the application must take into account aspects of design data access limited or low speed.
3. WAN data transfer level. As grid environments and distributed networks involve extensive grid any application must take into account issues of security, reliability and performance for handling data across the Internet or another WAN. Such applications require a logic performance for dealing with changing situations in which access to data may be slow or restricted.
4. Planning data transfer. Planning involves data following two main issues: ensuring the transfer of data to the appropriate location and time at which this transfer is required, taking into account the number and size of competing data to / from any resource.

Spurred the development of both high-speed networks and the increasing computing power of microprocessors, processing grid has a remarkable impact not only at academic level, but increasingly more and enterprise in all fields. Despite technological advances, however, less than 5% of Windows servers processing power and desktops, respectively 15-20% for UNIX servers is used. Companies make profits and offers exceptional rates, with a return period of low and low total cost compared to other technological solutions. [3]

Software implementations and middleware 2.3

- Advanced Resource Connector (NorduGrid's ARC)
- Altair PBS GridWorks
- Berkeley Open Infrastructure for Network Computing (BOINC)
- DIET
- Discovery Net
- European Middleware Initiative
- gLite
- Globus Toolkit
- GridWay
- InteGrade
- OurGrid
- Portable Batch System (PBS)
- Platform LSF
- ProActive
- Platform Symphony
- SDSC Storage resource broker (data grid)
- Simple Grid Protocol
- Sun Grid Engine
- Techila Grid
- **UNICORE**
- Univa Grid Engine
- Xgrid
- ZeroC ICE IceGrid

UNICORE 3

Figure 6UNICORE logo [7]

UNICORE (UNiform Interface to COmputing REsources) is a Grid computing technology that provides seamless, secure, and intuitive access to distributed Grid resources such as supercomputers or cluster systems and information stored in databases. UNICORE was developed in two projects funded by the German ministry for education and research (BMBF). In various European-funded projects UNICORE has evolved to a full-grown and well-tested Grid middleware system over the years. UNICORE is used in daily production at several supercomputer centers worldwide. Beyond this production usage, UNICORE serves as a solid basis in many European and international research projects. The UNICORE technology is open source under BSD license and available at SourceForge, where new releases are published on a regular basis.

In the late 90's of the 20th century the "ancestors" intended to create a uniform interface to computing resources for a (small) number of computing centers. Today - in the era of eSciences and large distributed eInfrastructures - UNICORE has become one of the most innovative and major middleware in Grid Computing serving users around the world [5].

UNICORE in Research 3.1

Many European and international research projects based their Grid software implementations on UNICORE, e.g. EUROGRID, GRIP, OpenMolGRID, VIOLA, or the Japanese NaReGI project. These projects extended or are extending the set of core UNICORE functions, including new features specific to their research or project focus. The goals of such projects were/are not only limited to the computer science community. Other scientific domains such as bioengineering or computational chemistry were / are also using UNICORE as the basis for their work and research, like in the OpenMolGRID or Chemomentum projects. Within the European DEISA project leading HPC centers in Europe joined to deploy and operate a pervasive, distributed, heterogeneous, multi-tera-scale supercomputing platform. UNICORE was used as the Grid middleware to access the DEISA resources.

UNICORE was deployed in distributed computing infrastructures in Europe (PRACE , the European Grid Infrastructure EGI) and was foreseen to be deployed in the upcoming XSEDE infrastructure in the United States.

UNICORE is successfully used in production environments, e.g. within the John von Neumann-Institute for Computing (NIC) to access the 294912 core "JUGENE" IBM BlueGene/P supercomputer and the 26304 core "JUROPA" cluster. The users of these resources come from a broad field of scientific domains including e.g. astrophysics , quantum physics, medicine, biology , computational chemistry , and climatology [6].

Here is incomplete list of UNICORE related projects :

1. UNICORE
2. UNICORE Plus
3. EUROGRID
4. D-Grid
5. GRIP
6. OpenMolGRID
7. UniGrids
8. VIOLA
9. DEISA
10. A-Ware
11. Chemomentum
12. OMII-Europe
13. EMI
14. NaReGI

UNICORE Security model 3.2

The security within UNICORE relies on the usage of permanent X.509 certificates issued by a trusted Certification Authority (CA). These certificates are used to provide a single sign-on in the UNICORE client, i.e. no further password requests are handed to the user. In addition the certificates are used for authentication and authorization, including the mapping of UNICORE user certificates to local accounts, e.g. Unix uid/gid, and for signing XML requests, which are sent over SSL based communication channels across 'insecure' internet links. Using X.509 certificates is one example for the consideration of well-known standards, e.g. released by the Global Grid Forum (GGF), within the UNICORE architecture. For trust delegation, UNICORE uses signed SAML assertions, while local authorization is controlled by XACML policies [7].

Architecture 3.3

From an end user's point of view, UNICORE is a client-server system which is based on a three-tier model:

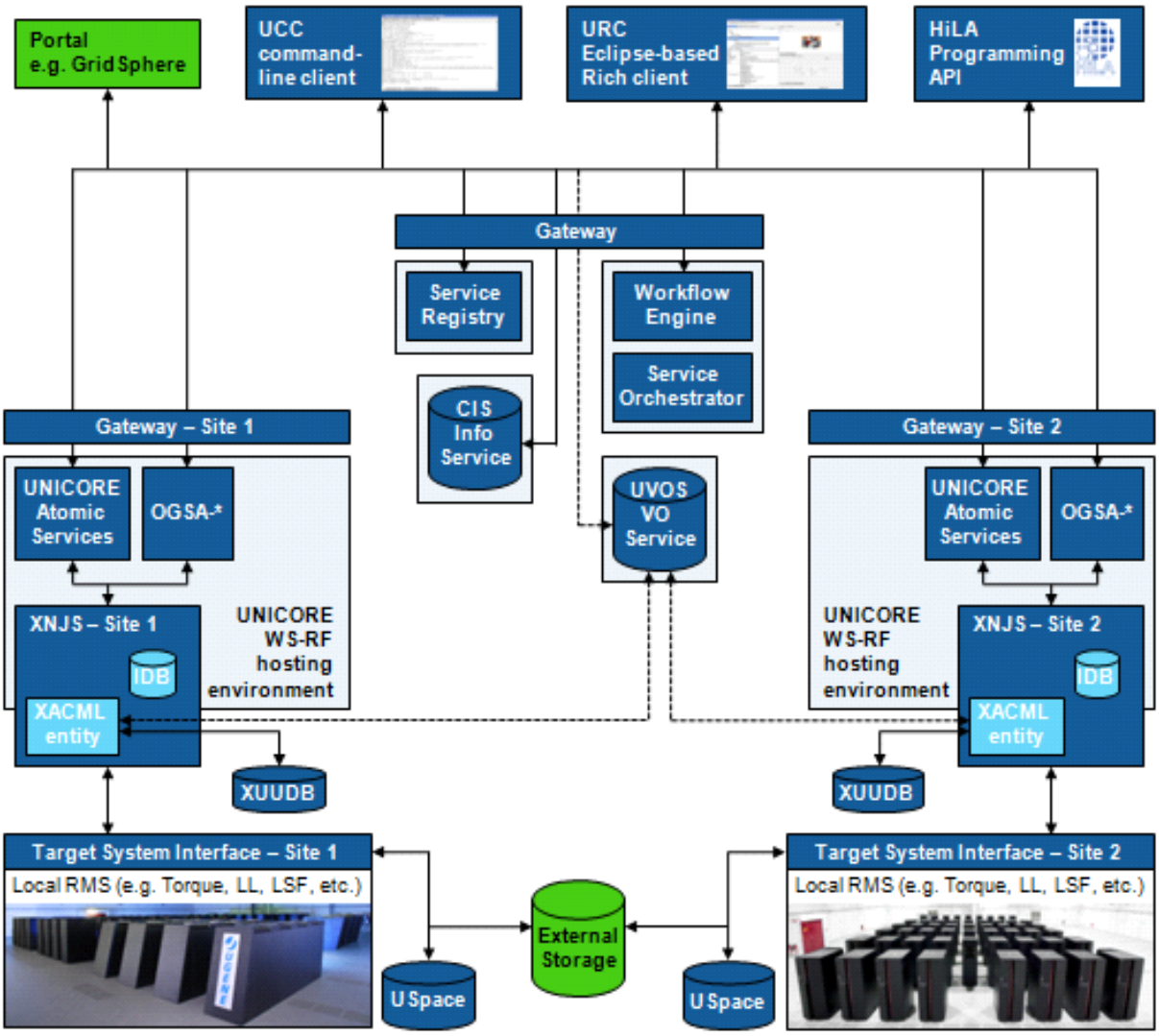


Figure 7 UNICORE architecture [8]

1. **Client Layer** - On the top layer a variety of clients is available to the users, ranging from the command-line interface named UCC , the Eclipse-based UNICORE Rich Client to the programming API named HiLA .

```
UCC version 1.2-SNAPSHOT
Usage: ucc <command> [OPTIONS] <args>
The following commands are available:
Data management:
ls                      - list a storage
copy-file-status        - check status of a copy-file
get-file                - get remote files
find                    - find files on storages
resolve                 - resolve remote location
copy-file               - copy remote files
c9m-get-file            - get remote files
put-file                - puts a local file to a remote server
General:
connect                 - connect to UNICORE
list-applications       - lists applications on target systems
list-jobs               - list your jobs
list-sites              - list remote sites
c9m-system-info         - Checks the availability of services.
Job execution:
run                     - run a job through UNICORE 6
get-status              - get job status
abort-job               - abort a job
batch                   - run ucc on a set of files
get-output              - get output files
Other:
shell                   - Starts an interactive UCC session
loadtest                - load tests services
issue-delegation        - Allows to issue a trust delegation assertion
wsrf                    - perform a WSRF operation
run-groovy              - run a Groovy script
Workflow:
c9m-submit              - submit a workflow to Chemomentum
c9m-trace               - trace info on a workflow in Chemomentum
c9m-control             - control a workflow in Chemomentum
c9m-workflow-info       - lists info on workflows in Chemomentum
Enter 'ucc <command> -h' for help on a particular command.
```

Figure 8 UNICORE Command line client [9]

2. **UCC** - The UNICORE command line client (UCC) is a very versatile command-line tool that allows users to access all features of the UNICORE service layer in a shell or scripting environment.
3. **UNICORE Rich Client** - The Eclipse-based UNICORE Rich Client (URC) offers the full set of functionalities to the users in a graphical representation. The same packages are responsible for visualising the output data of scientific simulations once the jobs have been executed and output files have been downloaded to the client machine.

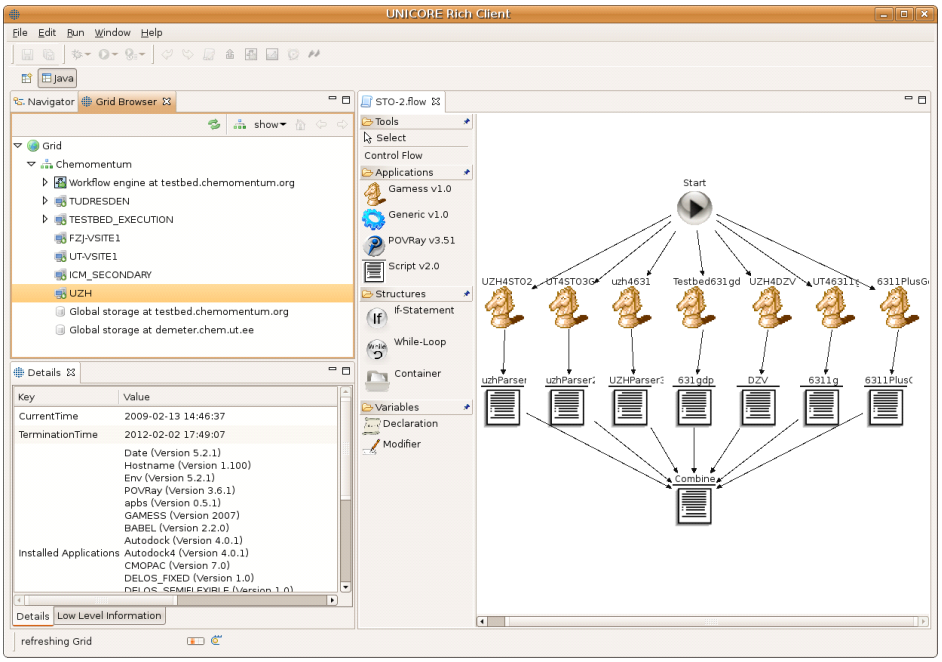


Figure 9 UNICORE Rich client [10]

4. **HiLA** - is a High Level API for Grid Applications that allows simple development of clients with just a few lines of code for otherwise complex functionality.

5. **Service Layer** - The middle tier or service layer comprises all services and components of the UNICORE Service-Oriented Architecture (SOA) based on WS-RF 1.2, SOAP, and WS-I standards.
6. **Gateway** - The Gateway component acts as the entry point to a UNICORE site and perform the authentication of all incoming requests. From the UNICORE perspective it is the "door to the outside world" in a site firewall and may serve several resources/target systems behind it.
7. **UNICORE/X** - The UNICORE/X server is the heart of a UNICORE site. Via WSRF compliant web services it provides access to storage resources, file transfer services and job submission and management services.
8. **IDB** - The IDB (Incarnation Data Base) is used during job incarnation, i.e. the mapping of the abstract job description (in JSDL) to the concrete job description for a specific resource (a shell script). Information about available applications and resource characteristics has to be defined in this database.
9. **XUUB** - The XUUDB user database provides a mapping from X.509 certificates to the actual users' logins and roles.
10. **UVOS** - As an alternative to the XUUDB, a Virtual Organisation (VO) service can be used to store user's attributes.
11. **Registry** - A single service registry is necessary to build-up and operates a distributed UNICORE infrastructure.
12. **CIS** - The Common Information Service (CIS) is the information service of UNICORE .
13. **Workflow** engine - The workflow engine deals with high-level workflow execution, offering a wide range of control constructs and other features.
14. **Service Orchestrator**- The service orchestrator layer is responsible to executing the individual tasks in a workflow, handling job execution and monitoring on the Grid.
15. **System Layer** - On the bottom system layer the TSI (Target System Interface) component is the interface between UNICORE and the individual resource management/batch system and operating system of the Grid resources.
16. **TSI** - In the TSI component the abstracted commands from the Grid are translated to system-specific commands, e.g. in the case of job submission, the specific commands like llsubmit or qsub of the local resource management system are called.
17. **Uspace** - The USpace is UNICOREs job directory.
18. **External storage** - For a transfer of data from and to external storage, the GridFTP transfer protocol can be used [8].

UNICORE Functions 3.4

As part of project UNICORE Plus a rich core set of functions has been developed that allow users to create and manage complex batch jobs that can be executed on different systems at different sites. The UNICORE software takes care of the necessary mapping of user request to system specific action. Some desirable functions have been deliberately excluded based on the expectation that they can be added in other projects. A brief summary of the key functions is given below.

Job creation and submission, Job management, Data management, Application support, Flow control, Meta-computing, Single sign-on, Support for legacy jobs, Resource management, High-performance file transfer, Distributed storage service, Metadata management: In order to make user data more accessible, an urgent need of services for organizing, indexing, and searching scientific data has to be satisfied. UNICORE was extended with a metadata service, which allows indexing, searching, creating and modifying metadata. This service is de-centralized, and stores metadata in a schema-free way as key-value pairs. It is using Apache Lucene as underlying indexing and search engine. The usage of the extensible metadata extractor framework Apache Tika which is integrated into the UNICORE metadata crawler allows for quick and easy processing of the scientific data [9].

METADATA 4

The term **metadata** refers to "data about data". The term is ambiguous, as it is used for two fundamentally different concepts . Structural metadata is about the design and specification of data structures and is more properly called "data about the containers of data". Descriptive metadata, on the other hand, is about individual instances of application data, the data content. In this case, a useful description would be "data about data content" or "content about content" thus metacontent. Descriptive, Guide and the National Information Standarts Organization concept of administrative metadata are all subtypes of meta content .

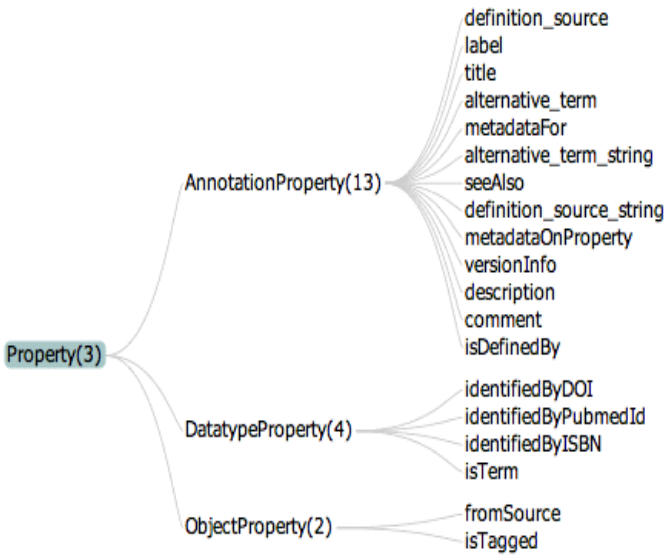


Figure 10 Metadata [11]

Metadata (meta content) are traditionally found in the card catalogs of libraries. As information has become increasingly digital, metadata are also used to describe digital data using metadata standarts specific to a particular discipline. By describing the contents and context of data files , the quality of the original data/files is greatly increased. For example, a web page may include metadata specifying what language it is written in, what tools were used to create it, and where to go for more on the subject, allowing browsers to automatically improve the experience of users.

Metadata (meta content) are defined as the data providing information about one or more aspects of the data, such as:

- Means of creation of the data
- Purpose of the data
- Time and date of creation
- Creator or author of the data
- Location on a computer network where the data were created
- Standards used

For example, a digital image may include metadata that describe how large the picture is, the color depth, the image resolution, when the image was created, and other data. A text document's metadata may contain information about how long the document is, who the author is, when the document was written, and a short summary of the document.

Metadata are data. As such, metadata can be stored and managed in a database, often called a Metadata registry or Metadata repository. However, without context and a point of reference, it might be impossible to identify metadata just by looking at them. For example: by itself, a database containing several numbers, all 13 digits long could be the results of calculations or a list of numbers to plug into an equation - without any other context, the numbers themselves can be perceived as the data. But if given the context that this database is a log of a book collection, those 13-digit numbers may now be identified as ISBNs - information that refers to the book, but is not itself the information within the book.

The term "metadata" was coined in 1968 by Philip Bagley, in his book "Extension of programming language concepts" where it is clear that he uses the term in the ISO 11179 "traditional" sense, which is "structural metadata" i.e. "data about the containers of data"; rather than the alternate sense "content about individual instances of data content" or meta content, the type of data usually found in library catalogues. Since then the fields of information management, information science, information technology, librarianship and GIS? Have widely adopted the term. In these fields the word *metadata* is defined as "data about data". While this is the generally accepted definition, various disciplines have adopted their own more specific explanation and uses of the term [11].

Metadata in UNICORE 4.1

UNICORE supports metadata management on a per-storage basis. This means, each storage instance (for example, the user's home, or a job working directory) has its own metadata management service instance.

Metadata management isseaprated into two parts: a front end (which is a web service) and a back end.

The front end service allows the user to manipulate and query metadata as well as manually trigger the metadata extraction process. The back end is the actual implementation of the metadata management, which is pluggable and can be exchanged by custom implementations.

Design 5

UNICORE includes a service for managing and searching metadata: the Metadata Management Framework. Within this framework it is possible to describe any given file stored in UNICORE storage by a set of user-defined attributes. One of the most important design decision made when implementing MMF, was to give the user maximal freedom as to what attributes she/he can use to describe the resources. MMF does not impose a rigid set of attributes that must be provided, but rather allows adding user-defined key-value pairs describing the content. In other words: no metadata schema is used. The current solution addresses the most general scenario of users representing different research communities working together on a common set of data. Each of such communities can use its own semantic to describe particular content and sometimes the descriptions are not compatible with each other. With the current implementation of the metadata manager, there will be no conflict between the descriptions. Such flexibility and freedom come at the cost of their own. The drawbacks of our approach are twofold. First, it is almost impossible to automatically validate the metadata provided by the user. Only syntax validation of the submitted metadata can be (and is) performed. Secondly, duplication and redundancy of the data can occur. We believe that the shortcomings are outweighed by the gained flexibility.

Further architecture decision made when developing Metadata Management Framework for UNICORE was to store metadata externally that is metadata will not be included in the original data files. Although some of the common file formats allow to extend files with additional attributes, e. g., it is possible to describe a jpeg image with Exif metadata, not each file format supports such an inclusions of the metadata. Thus we followed a different approach and stored the metadata externally in separate files. We also used a very simple syntax to store the metadata: JSON representation of the key-value pairs. The main criterion here was to avoid vendor lock-in, and make possible migration of the data (and corresponding metadata) as easy as possible.

Metadata implementation in UNICORE 6

All the methods for manipulating and searching the metadata are implemented as a standalone web service part of the UNICORE Atomic Services. It is a generic service and it theoretically allows to create and manage metadata descriptions for any given resource (not only files in a storage). However, the metadata support for storage resources was most urgently needed. Technically, the metadata service is implemented as a WS-RF web service. The metadata service and the storage access service (SMS) in UNICORE are closely coupled. For each UNICORE storage instance, a new instance of the metadata service is also created. Upon creation, the metadata service is provided with a reference to the storage so that the file access is realized exclusively via IStorage interface. An important ramification of such an implementation is the fact that the metadata service is compatible with all storage implementations available in UNICORE. The metadata service includes all basic methods for manipulating metadata, in form of a CRUD interface: CreateMetadata, GetMetadata, UpdateMetadata, and DeleteMetadata . When creating or updating a metadata description of a file stored in a UNICORE storage, the user has to provide a file name and a set of key-value pairs. The description is stored in JSON format in a file fileName.metadata , where fileName is the file name of the described file. For deletion or retrieval of the metadata only a file name has to be provided. Since the manual creation of the metadata is an error-prone and cumbersome activity, we provide a tool to support the process. The user can start automatic extraction of the metadata in a given storage, by calling the StartMetadataExtraction method of the metadata manger service. The service will then go through all the files in the given storage directory and try to extract metadata automatically by using the Apache Tika content analysis toolkit4. Tika supports a wide range of file formats, e. g., xml, doc/odt, pdf, jpg and it is able to extract a lot of useful information [8].

It can also be easily extended to support further formats. Note that such an extraction needs to be performed only once for the storage and not each time a user wishes to access the metadata.

Given the metadata describing files are available (as a result of either manual creation or automatic extraction), the searching functions of the metadata service can be employed. The respective web service function name is SearchMetadata and it takes two parameters: a query string and a Boolean value indicating whether to perform an advanced query or not. The functionality is realized with help of powerful search engine: Apache Lucene. Each of the metadata manipulating methods described above, besides storing the data in the storage, also passes the metadata to the index to keep it up-to-date. While searching, the query string is passed to Lucene and returned matching items are passed back to the user.

Lucene supports advanced query syntax e. g., wildcard queries, range queries and compound queries. Probably the most interesting query type supported by Lucene are the “fuzzy” searches, where not only exact matches but also all items semantically similar to the query string (ordered by the similarity grad) are returned. A quick overview of supported query types is shown below, for more examples look to the documentation of the Apache Lucene project.

Query type	Syntax example	Result
Wildcard	foo*	foot, foobar ...

Range	[bar TO foo]	bar, center, desk ...foo
Compound	foo OR bar	foo, bar
Fuzzy	foobar	foobar, flobar, foober…..

Presented methods of the metadata service embrace most of the basic functionality a user might expect, including advanced searching and automatic metadata extraction. They probably do not cover all possible use-cases of metadata manipulation, but strike a good balance between simplicity and sophistication [10].

Why it is important 7

For convenient data management especially when data amount is too large search engine is required. It is foreseeable that a considerable amount of data and metadata will be produced in such grid systems as UNICORE. Depending on the type of executed workflow and the level of detail, the size of metadata can reach hundreds of kilobytes, the data even up to gigabytes.

Therefore a highly scalable, distributed and decentralized Grid search engine is needed. The storage of data and augmenting metadata as search engine is the fundamental requirement for such distributed systems. An important feature is the possibility to define search criteria, which provides convenient data filtering. Clearly defined and well structured metadata federated search engine is crucial to provide adaptive user interfaces for diverse purposes. As UNICORE is a grid platform consequently data and their metadata are distributed between several storage consequently federated type search engine, which collects data from different media and returns summarized, decorated with additional useful information, is a logical approach.

Significant of research and comparison with other grid platforms 8

There is no too much grid platforms (due to the complexity of their implementation) – especially platforms supporting metadata. As it was maintained above in case of distributed calculation and data storing - federate data search is a fundamental requirement for such type systems. We can not say that this is unique idea or unique approach - it is logical approach for grid systems.

Implementation of this approach is unique for each platform, due to unique specification of architecture. But main conception - idea that each node should return suitable data to the requester - is same.

Here I want to describe federated search engine of two platforms / plug-ins which support metadata management.

MCAT 8.1

MCAT is a meta information catalog system implemented at SDSC as part of the Data Intensive Computing Environment with requirements mainly based on the Storage Resource Broker (SRB) system. MCAT catalog is designed to serve both a core-level of meta information and domain-dependent meta information. MCAT Version 1.1 was released along with SRB Version 1.1 in March 1998.

I briefly describe the MCAT architecture along with the view about how it can interact with other catalogs and/or support extensible (read application dependent) meta information. Figure below provides architecture for MCAT system which provides an extensible architecture.

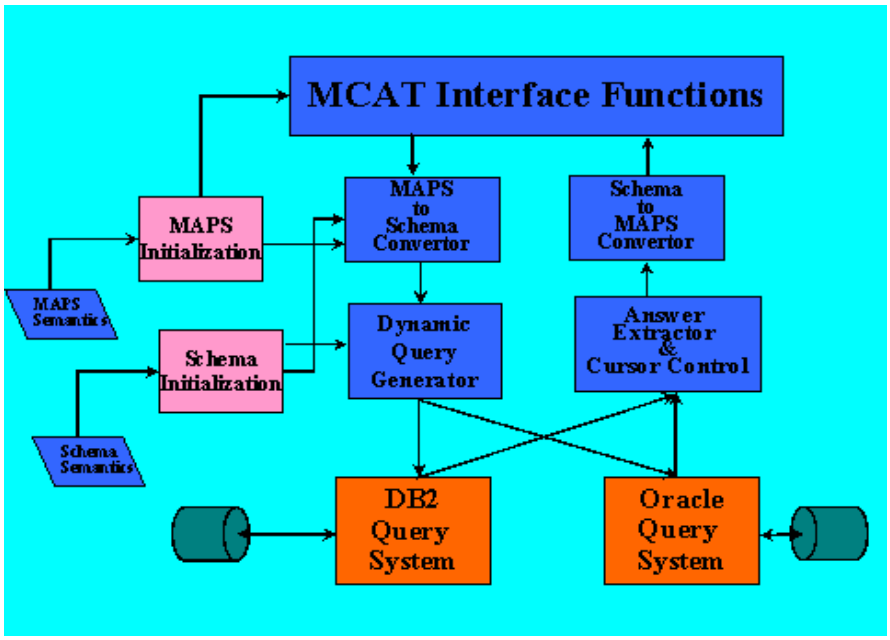


Figure 11 MCAT Architecture [12]

As can be seen from the architecture defined, MCAT provides an interface protocol for the application to interact with MCAT. The protocol uses a data structure for the interchange which is called MAPS - Metadata Attribute Presentation Structure. The data structure (which will have a wire-format for communication and a data format for computation) would provide an extensible model for communicating metadata information. Internal to MCAT, the schema for storing metadata might possibly differ from MAPS (eg. a database schemata may be used), and hence mappings between internal format and MAPS would be required. Also, the data interchanged using MAPS is identified through queries to the Catalog and hence a query generation mechanism is used in the process. Moreover, the query generator would use internal schema definitions, relationships and semantics to develop an appropriate query. Depending upon the catalog, different types of query would be generated. For example, if the catalog resides in a database, then SQL would be the target language of the query generator; if the catalog is in LDAP, then LDAP - specific query would be produced. This query generation in its extensions would be able to deal with multiple catalogs as well as heterogeneous catalogs, both internal and external to MCAT. Figure below provides the components of a system that can handle multiple catalogs both internal to MCAT and external to MCAT.

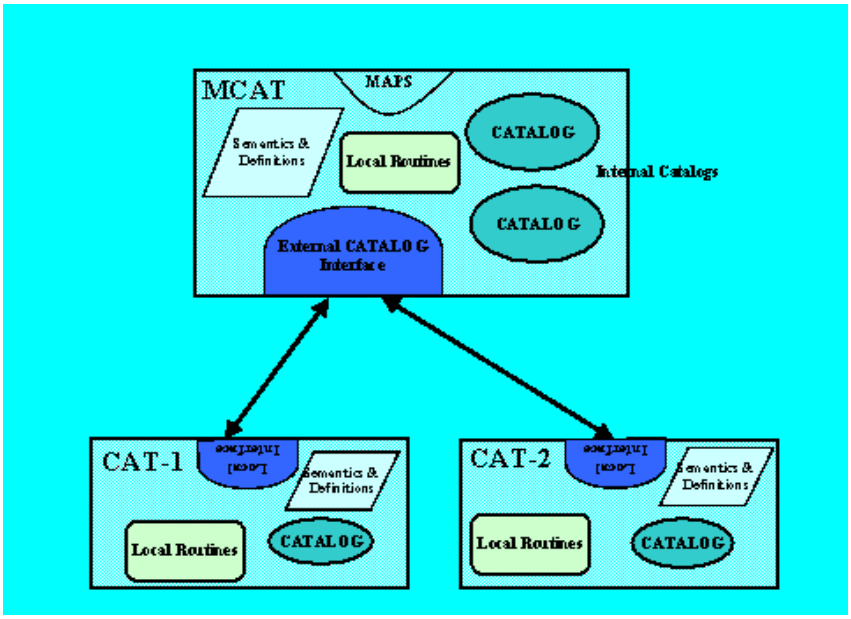


Figure 12 MCAT [13]

Figure above provides a mechanism for multiple catalogs to be served through MCAT. Assume that a group has their own database with a large quantity of metadata. Moreover, the organization might be peculiar to the group's field of activity and the metadata may reside as different types of objects - eg. files, tables, etc. The MCAT- to-catalog interaction is facilitated by a uniform abstract interface (to be define and may be called the Catalog Interface Definition) that allows external catalogs to communicate with MCAT. The communications would be of two different types: meta - metadata communication wherein semantics of metadata in the external catalog is communicated to MCAT and metadata communication where metadata is transferred between the two catalogs in response to queries and updates. With the definition of an abstraction, including new catalogs would become an exercise in writing middle ware components (similar to what is done in the case of media-drivers in SRB.)

A system of MCAT zones or Federation has been implemented. Each MCAT manages metadata independently of other zones. There was no change for most metadata and metadata operations. New metadata includes: Zone Info - new table defines all zones in the federation, ZoneName - logical name, Network address, Local/Foreign flag, and Authentication information. Each MCAT maintains a complete zone table.

The MCAT includes user information which defines all users in the federation. There is a single global user name space, so each username@domain name must be unique in a Zone federation. Each MCAT maintains a table of all users, with a flag to indicate if the user is local or foreign. A user is local to only one zone. Sensitive info (the user's password) is only stored in user's local zone.

MCAT more looks like plug-in, which can be installed separately and not system integrated tool.

In fact MCAT system integrates several machines and becomes central metadata manager, in federated version there is same architecture but it enabled interaction between several MCATs to exchange meta information [12].

AMGA 8.2

AMGA - The ARDA Metadata Grid Application is a general purpose metadata catalogue and part of the European Middleware Initiative middleware distribution. It was originally developed

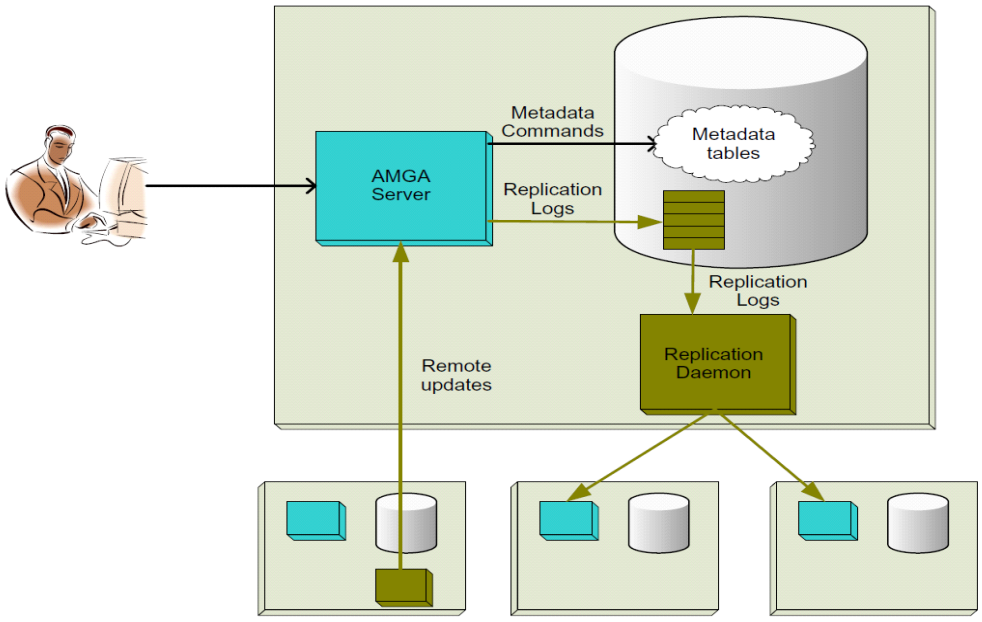


Figure 13 AMGA replication [14]

by the EGEE project as part of its gLite middleware, when it became clear that many Grid applications needed metadata information on files and to organize a work-flow. AMGA is now developed and supported by the European Middleware Initiative.

AMGA as a metadata service allows users to attach metadata information to files stored on the Grid, where metadata can be any relationally organized data typically stored in a relational database system. In addition, the metadata in AMGA can also be stored independently of any associated files, which allows AMGA to be used as a general access tool to relational databases on the Grid. AMGA features a simple to learn metadata access language, which has been very useful for the adoption of AMGA in smaller Grid applications, as it considerably lowers the technical hurdle to make use of relational data.

One of the main features of AMGA, and one unique to it, is the possibility to replicate metadata between different AMGA instances allowing the federation of metadata , but also to increase the scalability and improve the access times on a globally deployed Grid (as done by the Wisdom project). Performance and efficiency of the access across WANs has been independently targeted by an access protocol optimized for the bulk transfer of metadata across WANs using data streaming [13].

Replication allows a catalogue (slave) to create a replica of part or of the totality of the metadata contained in another catalogue (master). The system ensures that the replica is kept up-to-date, by receiving updates from the master. The clients of the slave catalogue can then access the replicated metadata locally. AMGA uses an asynchronous, master-slave model for replication. The use of asynchronous replication is motivated mainly by the high latency of wide-area networks, where synchronous replication does not scale properly. Figure below presents the replication architecture of AMGA. The basic mechanism is to have the master keep on its local database a log of all the updates it performed on its back-end. [14]

The federation mechanism in AMGA provides a user a virtualized view on metadata as if one metadata server has all data which are actually distributed at multiple sites. Federation in AMGA is very similar to mounting in NFS. That is, if a remote directory is mounted to a local directory, a user may access it as it is located at local site. Federation actually redirects user's query to other multiple AMGA nodes and integrates the results. The list of other AMGA nodes to redirect user's

query should be set properly on a per-directory basis. AMGA provides two types of federation methods:

1. Server-side federation
2. Client-side federation.

In server-side federation, an AMGA server does the actual federation, which redirects a query and integrates results.

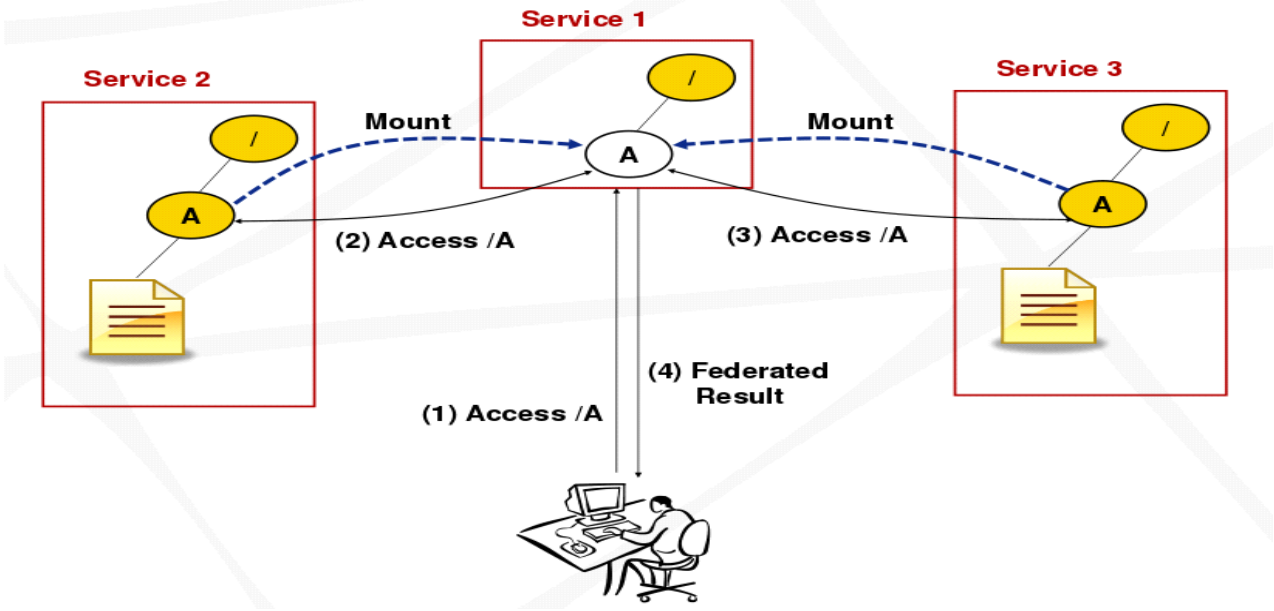


Figure 14 AMGA Server side federation [15]

Meanwhile, in client-side federation, an AMGA server provides a client with a list of AMGA nodes to redirect user's query and a client side does the actual federation. The client-side federation puts little overhead on the AMGA server caused by federation. The server-side federation may have high overhead caused by federation at the server-side.

It is necessary to have one **central** AMGA server which maintains all the information about federation. That is, it needs to be configured with the settings of other AMGA nodes that it might be required to contact for federation.

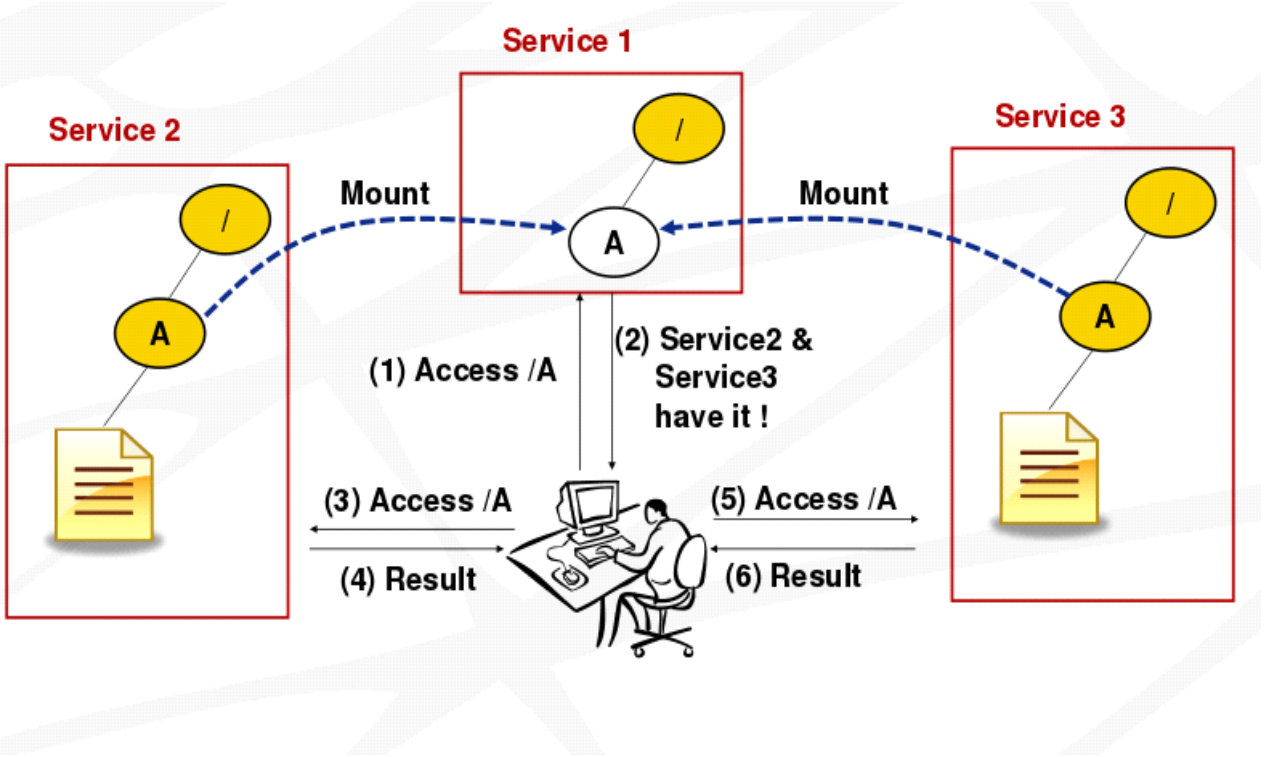


Figure 15 AMGA Client side federation [16]

In federation, connections to other AMGA nodes use the same protocol as the ones from clients to an AMGA server, including the support for authentication and encryption. With the client-side federation, all the necessary configuration settings should be stored at the mdclient.config file. And with the server-side federation, site information should hold necessary configuration settings (e.g., use SSL, authenticate with password, certificates or grid proxies,...) [15].

Basic requirement 9

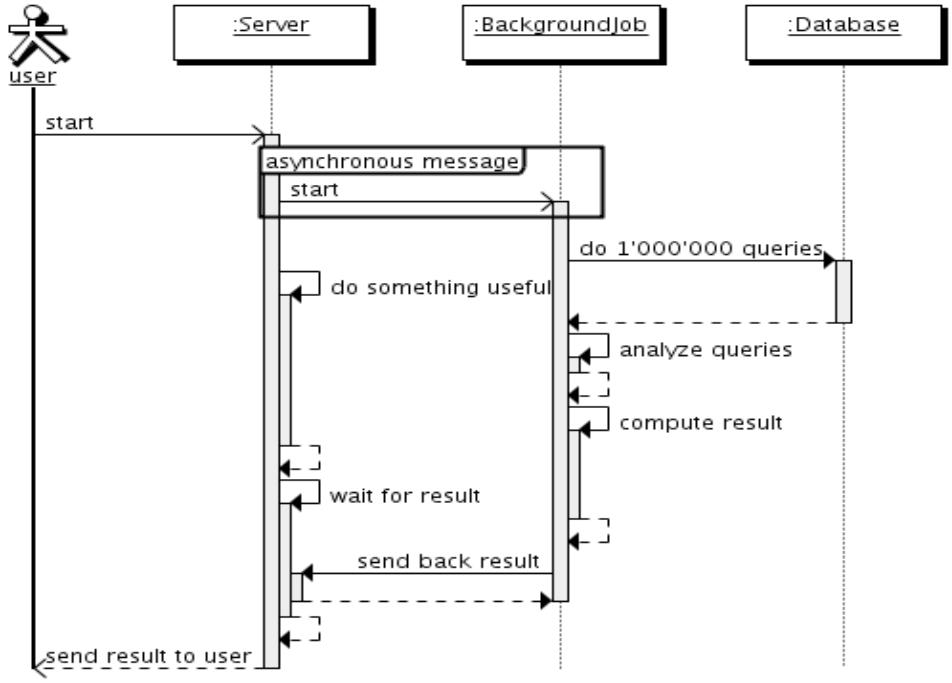


Figure 16 User interaction with system [17]

UNICORE is a distributive - grid platform, consequently it stores data on different nodes/media. For effective data management, which are distributed in different storages, it is necessary search engine. Despite the fact that search will be federated, user should be able to run single query and as a result get, collected from different storages, single list. List should be organized that way, to clarify which results belong to which storages and also some useful statistical information – count of processed storages, time of search etc.

Federated search execution time depends on count of storages it process, in case of large number of storages may it take too much time, so it was decided that search should be asynchronous. In this case asynchronous means after user fire search, system won't be in standby mode – it will be able to process user's other operations, while search will be running in background. After search engine finish its work, user will be notified that search result are ready, also user can log out after firing search and at the moment of next authentication he/she will get prepared search result .

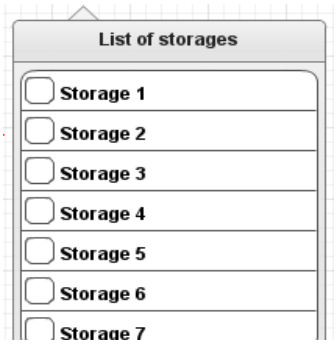


Figure 17 List of storages

In case of asynchronous search, which may take a lot of time it is advisable to add future for user to check search state, at any time of search process.

Because grid in itself may contain a lot of nodes, it may be inexpedient to provide search in all nodes, so it was decided that user should be able as criteria to specify in which storages search should be processed.

Basic requirements: For effective data management in "UNICORE", it should be added federated search tool. This tool should work asynchronous; user should be able to specify list of storages where search should be provided and at any time of search process user should be able to check its status/state.

General actions sequence 9.1

Picture below illustrates general user actions sequence with UML use cases. Independently of the implementation model of the problem; the general scenario of user and system interaction is same.

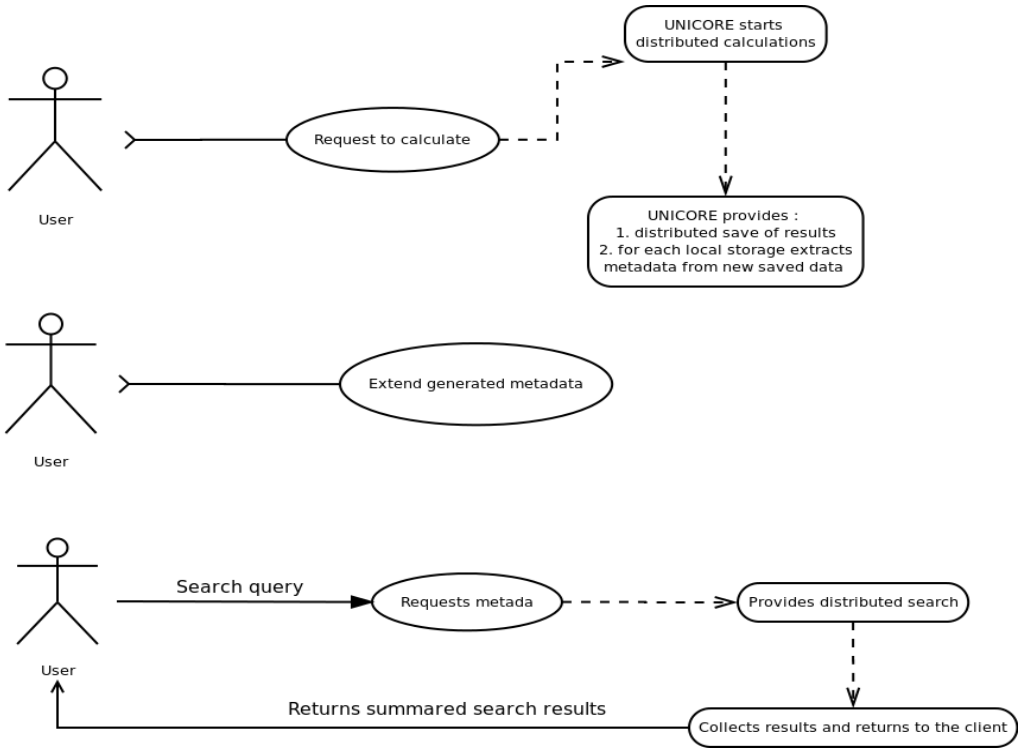


Figure 18 Possible user's actions

As it shown in diagram's first row - first user connects to the system and sends request for calculations or some other commands as a result of system generates some type of files. System accepts request and processes it. After certain operations on the basis of the results system generates some files. Further there is a persistence of these results on different media, after that system automatically starts metadata extraction of new saved object via Apache Tika and saves them in separate file with *.metadata extension in JSON format. .

Further it is possible next scenario: user can send request to system to extend metadata in "key – value pair" format to describe or specify some properties of object which this metadata is referring – this process is shown on in the second row of diagram above.

After a while user may need information about some data. User requests metadata using search engine, specifying different parameters like data carrier (grid node URL) , creation date, during which experiment / calculation object it refers was generated / created and etc. To do this user sends to the system search request with specified criteria. System starts federated search (currently no matter what

kind federated search design it will be), after receiving a response from each carrier it summarizes them and sends to the user – the third row of UML diagram.

General scenario of user and system interaction should look something like this.

Short description of patterns of diagrams 10

For clearer illustration of different type architecture, it was decided that all architectures should be shown for on grid configuration.

Under the configuration is meant to be used a certain number of interactive user and system, a certain amount media / servers, a certain number of clients and certain amount of grid nodes. There is used one configuration for all architectures:

- One client and seven grid nodes
- One supercomputer
- One server
- Five casual PCs connected to Grid.

Also for some dioramas there are illustrated indexers, they needed to show some specific architectures

Each supremely listed object have their own figure all the figures are figures of Uml standards Sybase, Cisco network, Cisco computers:

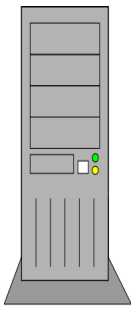
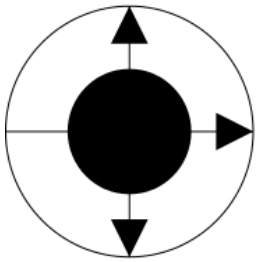


Illustration : Server



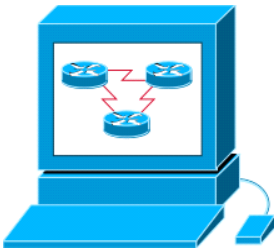
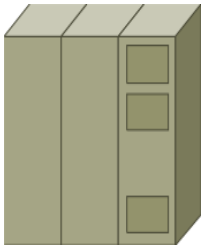
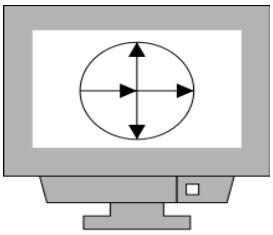
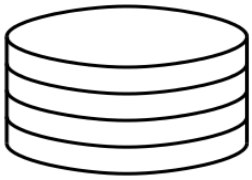


Illustration : Replication service

Illustration : Some usual computer in Grid







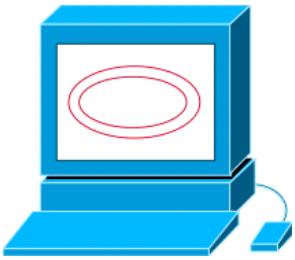


Illustration : Supercomputer

Illustration : Distribution server

Illustration : Index storage/server it can be Lucene or any RMDBS

Possible approaches 11

In this section it will be presented several approaches of solving "Federated metadata search" problem with different level of efficiency. It will be described conception of each approach and its architectural design, pros and cons of them. In particular it would be described 4 approaches:

1. "Single request and single response"
2. "Multi requests – multi responses - (chosen variant)"
3. "Approach with special/central distributor server"
4. "Approach with special/central indexer server"

Each approach will be described briefly, with implementation requirements / stages, UML diagrams and purposes why it was declined/chosen.

But first it will be described general model of user/system actions.

Central distributor architecture 11.1

The main future of this architecture is that in each grid infrastructure besides grid nodes has specialized distributor server.

Distributor is a server which designed to process search queries, distribute between competent grid nodes, collect results from grid nodes and return back to the client which initiated federated search

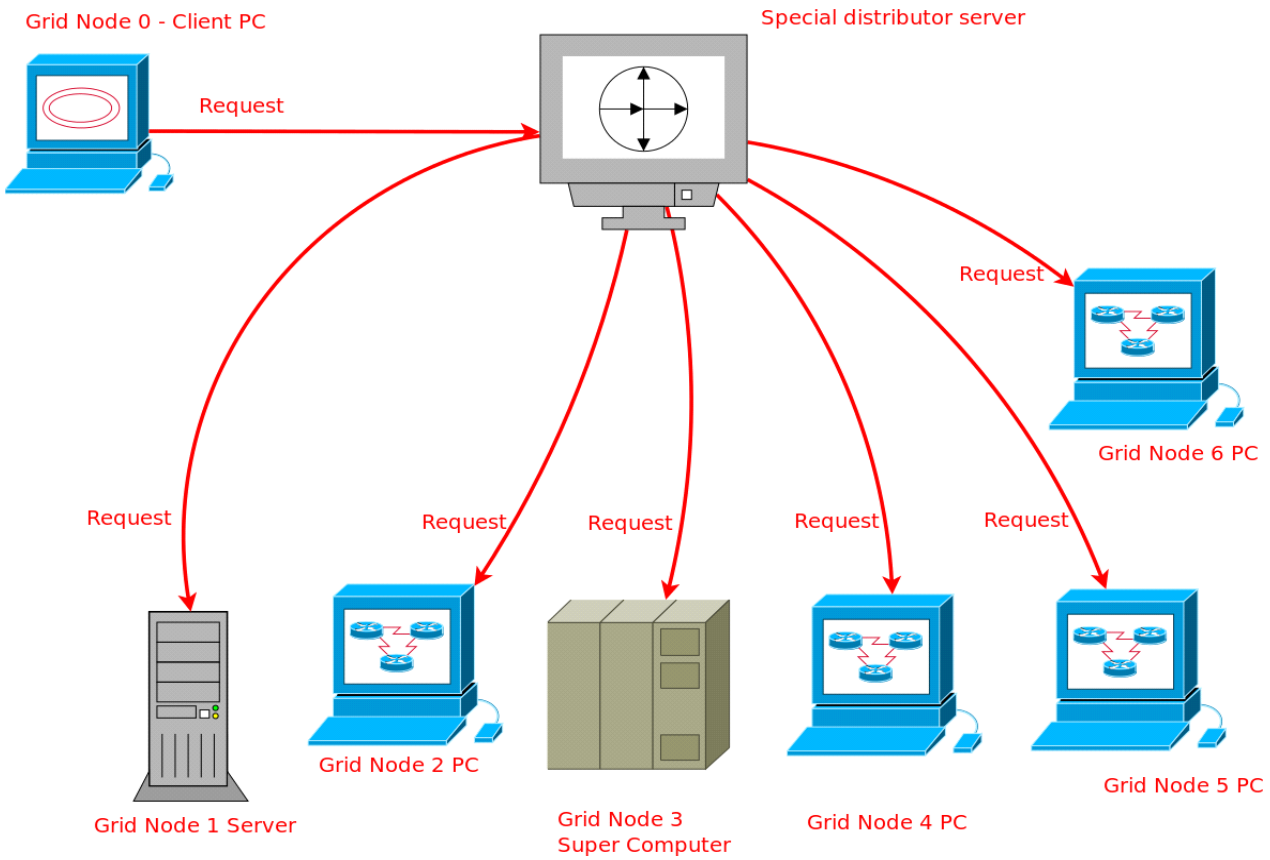


Figure 19 Central distributor architecture

The principle of this architecture is shown in the figure above: Client (in the figure "Grid Node 0") sends search request to the distributor server. To implement this scenario, client had to send a request with the following parameters:

1. It's ID (for distributor to know who return the data found).
2. List of grid nodes where search should be processed.
3. Search Query

Due to the fact that it can be connected too much servers in grid infrastructure and that single search request processing can take a long time – system with building client requests in a priority queue and stay distributor in standby mode is not acceptable for such type of architecture. Therefore a distributor must operate asynchronously - at any time to be able to receive search requests from any clients.

After receiving request distributor generates ID for it – each search request must have unique ID, after that it makes "key value pair" with client ID and generated search request ID, where request ID is key and client ID is value. This kind of mapping is necessary for following causes:

1. Because distributor has asynchronous architecture it could receive several search requests at same time, during processing one, even from same client machine with different search criteria. In order to not confuse them it has to mark them with unique IDs.

2. After receiving answer/result from nodes and packaging them distributor should know to whom it should return data and define for which search session this request is.
3. It is also necessary for logging in cause of system fails.

After receiving search request and conduct all necessary operation it distributes query, with parameters described above, between requested nodes.

The second stage – response stage of the architecture is shown on the diagram below :

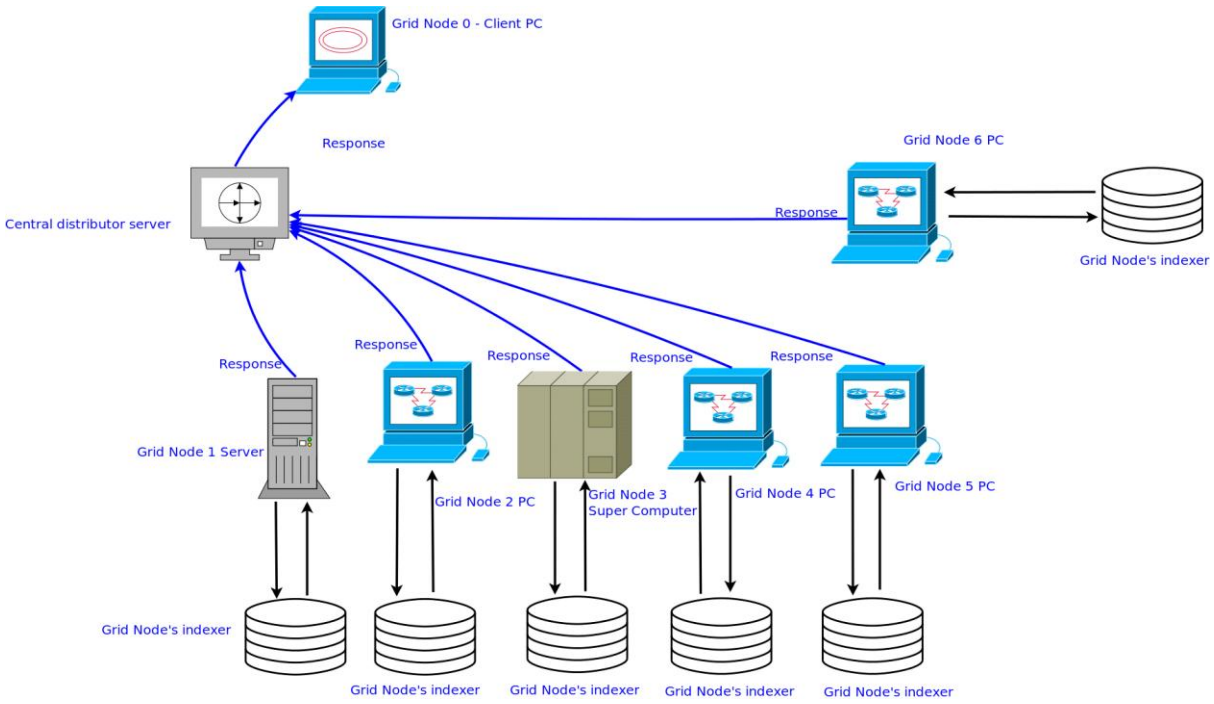


Figure 20 Central distributor architecture 2

Each node receives search query from distributor and sends it to its Indexing system (it not necessary that it should be Lucene - it could be some RDBMS). After receiving the results satisfy the search query grid node sends to the distributor them together with the ID received via parameters. Distributor after getting response from node starts to search ID matches in its query distributing map. If all nodes from the map have sent response then it wraps them and sends as response back to the client.

The advantages of the above-described architecture are:

1. Easy implementation
2. Doesn't require installation of special distribution module on each node, just on the distributor server.
3. Fast query processing – server which concentrating only on query distributing will work faster than servers combine this functionality.
4. There is two disadvantages of this design:
5. In case of failure of the central distributor, fails all search engine – grid stays without without search tool.

6. It is fundamentally contrary to the philosophy / architecture UNICORE – UNICORE is open source grid platform, any interested can download and make own grid infrastructure. Implementing federated search this way we forcing the user to purchase a separate server for distribution and install a separate software implements distribution.

This architecture was expelled in the beginning of review.

Single request single response 11.2

As can be seen in the title the main concept of this architecture is one request one response. Here appeal is that the architecture does not require additional server to distribute search queries between grid nodes

The architecture works as shown on the diagram below:

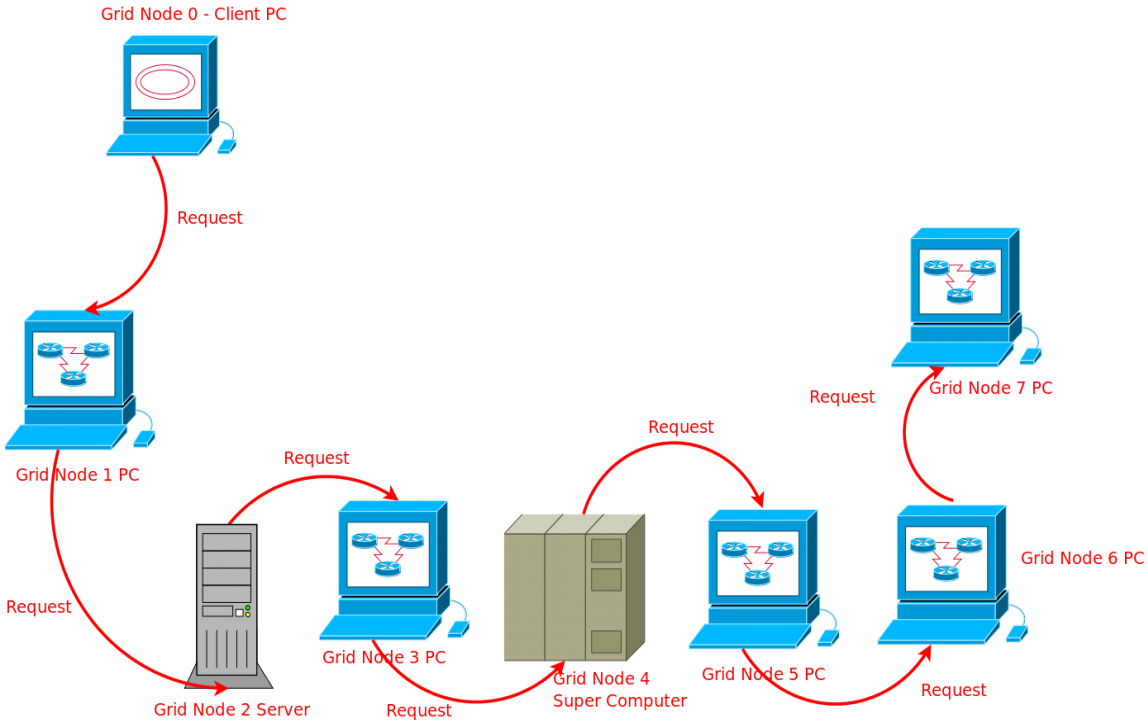


Figure 21 Single request single response

On the first stage client sends search request to one of the grid nodes where search should occur with list of other grid nodes. Client also should send to the neighbor grid node next parameters :

1. Its ID (in order to be possible reverse data transfer)
2. List of server where search should be processed
3. Search query
4. Search request session ID

Node which received request extracts its ID from list of servers , conducts search in its system and asynchronously sends search request with same parameters to the node chosen from list, but instead of client id as receiver sends its own ID, to receive results from that node. This way each node sends search

request to another while there is any item in the list of nodes where search should be processed . Sending IDs guarantee, back chain construction.

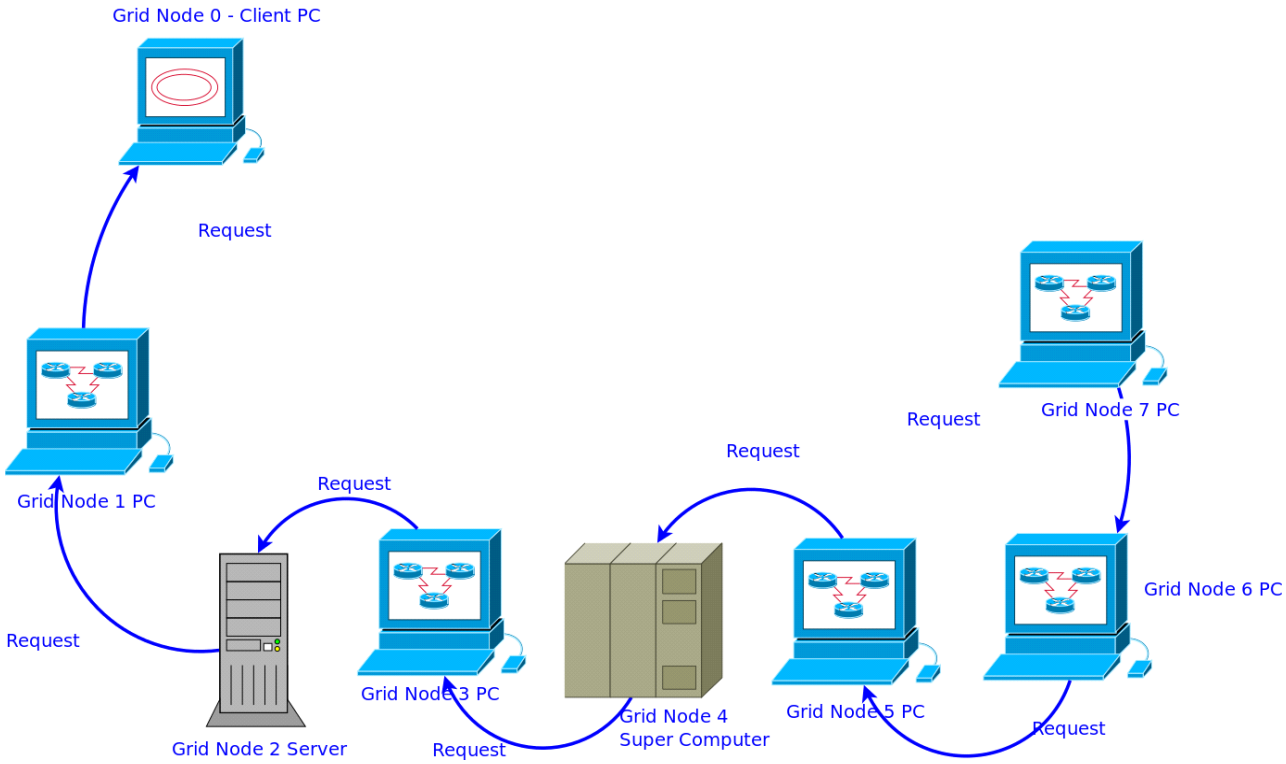


Figure 22 Single request single response 2

In the last step when last node in the list received request (with empty node's list) this node conducts search locally , gets results and sends back to the node which sent him request. That node gets response merges with its own and sends assembled results back to the node which called it – and so until all results are back to the client node started search.

Advantages:

1. The only advantage of this architecture is that there is one request and one response
2. Disadvantages :
3. Each node has to wait other to send response back.
4. The main disadvantage is that when the chain is broke (one of the grid nodes failures) collect backward chain is impossible – search fails.
5. Difficulty of implementation - each node would have to have two futures be called and call other nodes.
6. If one of the nodes is process search slowly all the search session will be slow.
7. Potential slowness – cause of the data transfer between nodes in network.

This approach was expelled cause of above listed purposes and difficulty of its implementation.

Approach with special/central indexer server 11.3

This approach conducts one of the fastest searches, but this speed requires long preliminary work.

In grid infrastructure should be one separated server designed only for data indexation. All search requests should be processed through this indexation server

Consider the principle of this design step by step:

In the first stage client sends request to the indexation server with following parameters:

1. Search query
2. List of nodes
3. It's ID.

Client ID is required because of asynchronicity of indexation server to log search requests sessions and send back results to caller.

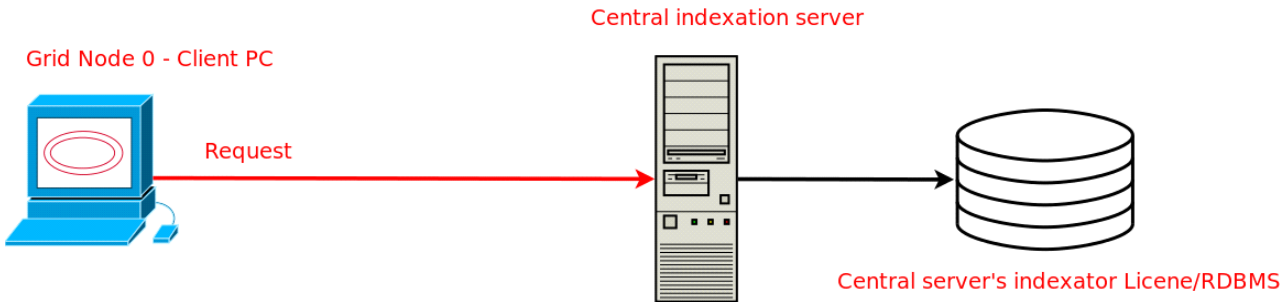


Figure 23 Central indexer server

Central indexation server gets request and processes search in its indexation system.

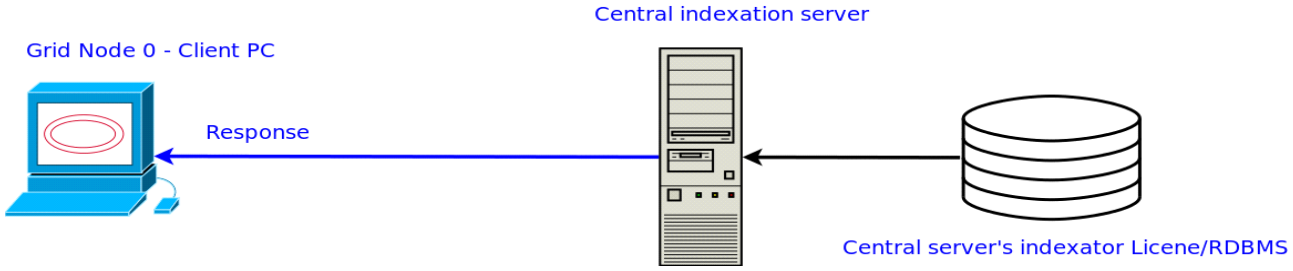


Figure 24 Central indexer server 2

Finding the answer sends the requested data back to the client. This design is simple, easy to implement and works fast. But there is question how to anticipatorily index metadata from different nodes. Here could be two approaches:

•Indexation server has to crawl all grid nodes in some time interval.

•Each grid node after created/updated metadata file has to report to indexation server that changes have been occurred.

First variant is pretty easy but has one big disadvantage : – time window when there is metadata exists but server haven't crawled node where it is located, so client sends request to find existing metadata and server responses that it doesn't found.

Therefore it would be appropriate to consider the second option – method with auto replication of metadata in case of changes. Although in this case we would have to write a special replication utility for both the server and for the grid nodes. The replication option would look like this:

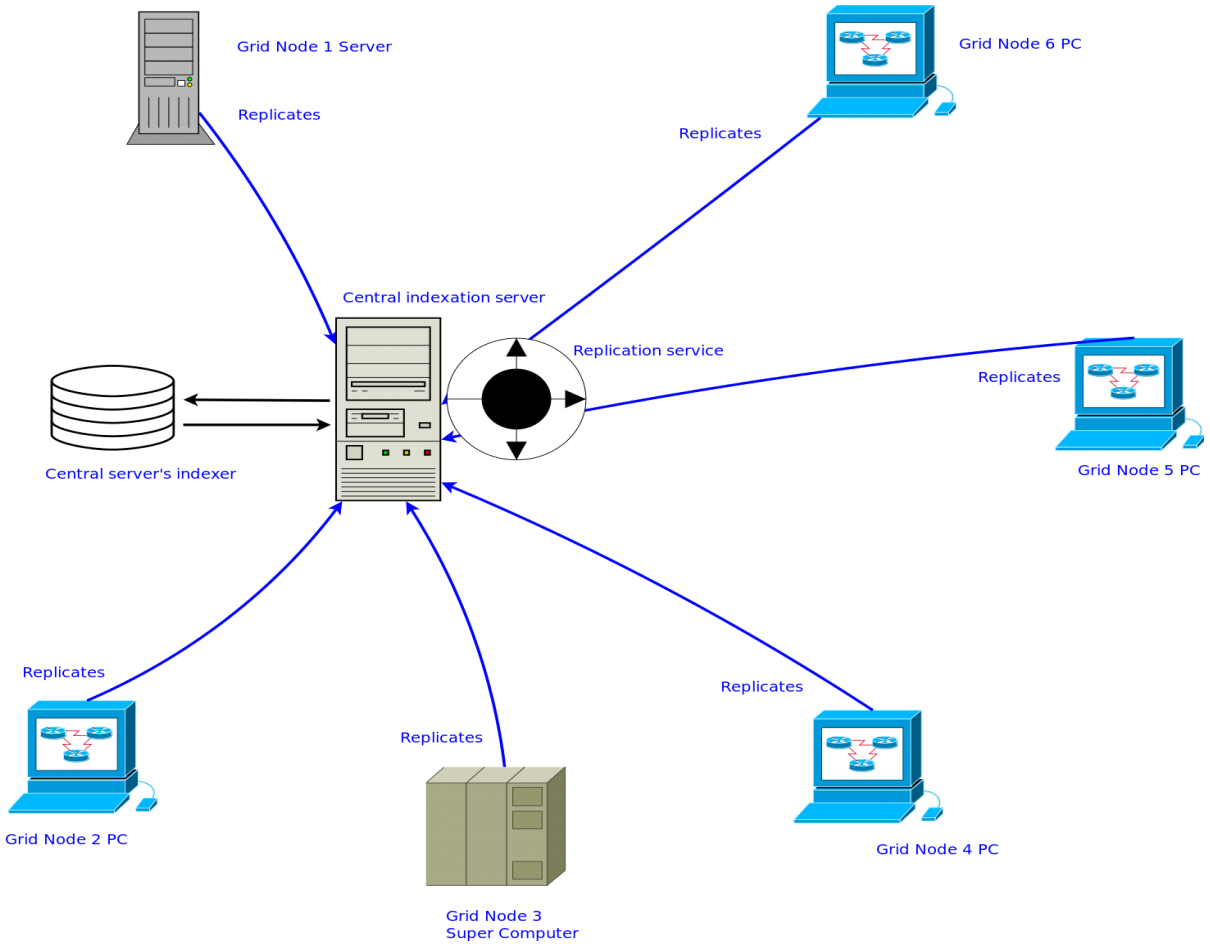


Figure 25 Central indexer server 3

On the picture above its shown how each node calls replication service and sends new version of metadata. Although it sends to which file refer this metadata for server to know which record should be updated. Advantages:

1. Too fast.
2. Disadvantages :
3. In case of failure of the central indexation server, fails all search engine – grid stays without search tool.
4. It is fundamentally contrary to the philosophy / architecture UNICORE – UNICORE is open source grid platform , any interested can download and make own grid infrastructure. Implementing federated search this way we forcing the user to purchase a separate server for indexation and install a separate software implements indexation logic.

5. Replication is too time expensive operation, which busy both server and grid node which replicates data.

Multi requests and multi responses 11.4

This design was chosen for federated metadata search in UNICORE. The main idea of this architecture is : client sends search request to the grid nodes which it interested in and each nodes independently sends back result to client. Client defines:

•On which machines connected in grid infrastructure should search be processed

•Search query.

Work stages of this approach looks like shown below:

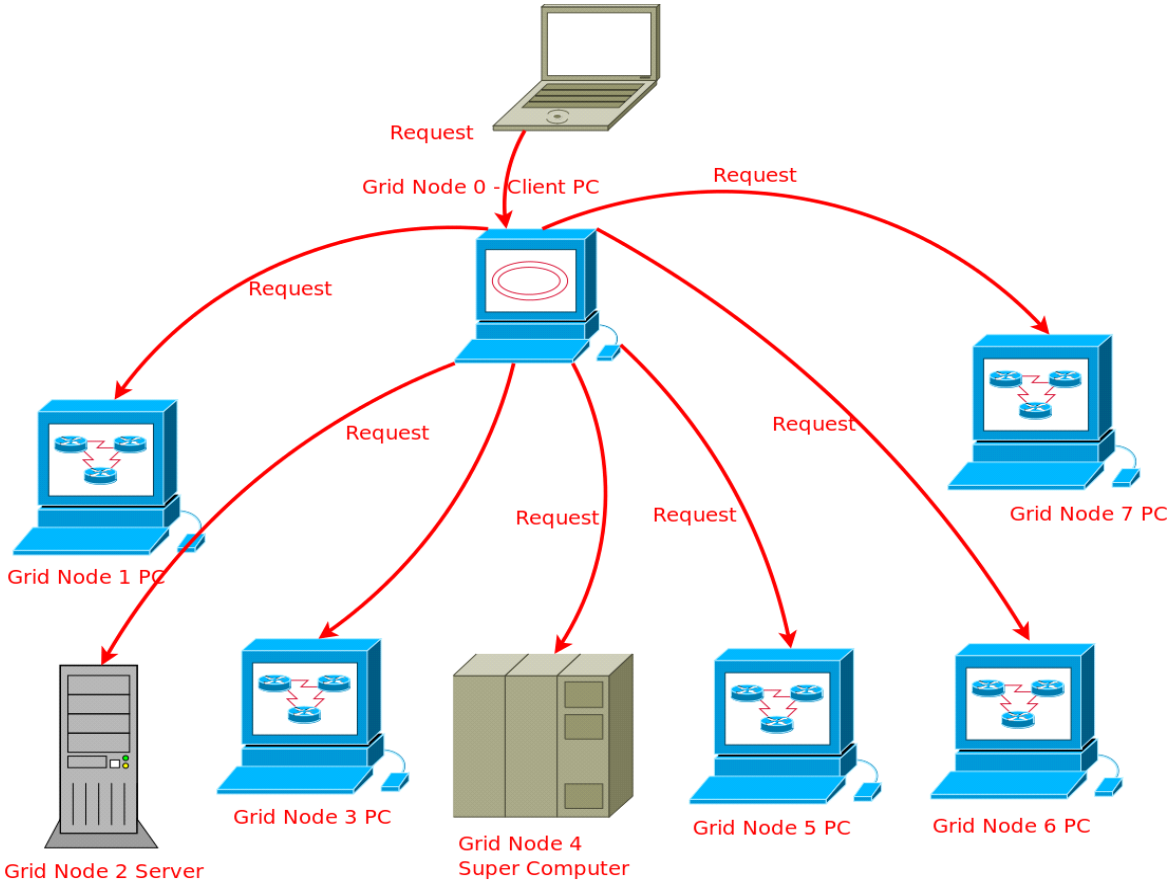


Figure 26 Multi requests and multi responses

Client (any grid node can be client or some other PC which is connected to grid but isn't node – it connects to some grid node) starts asynchronous procedure and iteratively sends request for search to each node from interest list, also to itself. Search request contains only search query. Asynchronous procedure works in request response mode – what means client doesn't iterate to another node until it get answer from node or define that node is not corresponding .

In this scenario there is no risk that cause of some grid node's failure search process could be fail, even in case of failure of all grid nodes from list search can be considered as valid. Procedure can be integrated in UNICORE, so that user doesn't have to install any special plug-in.

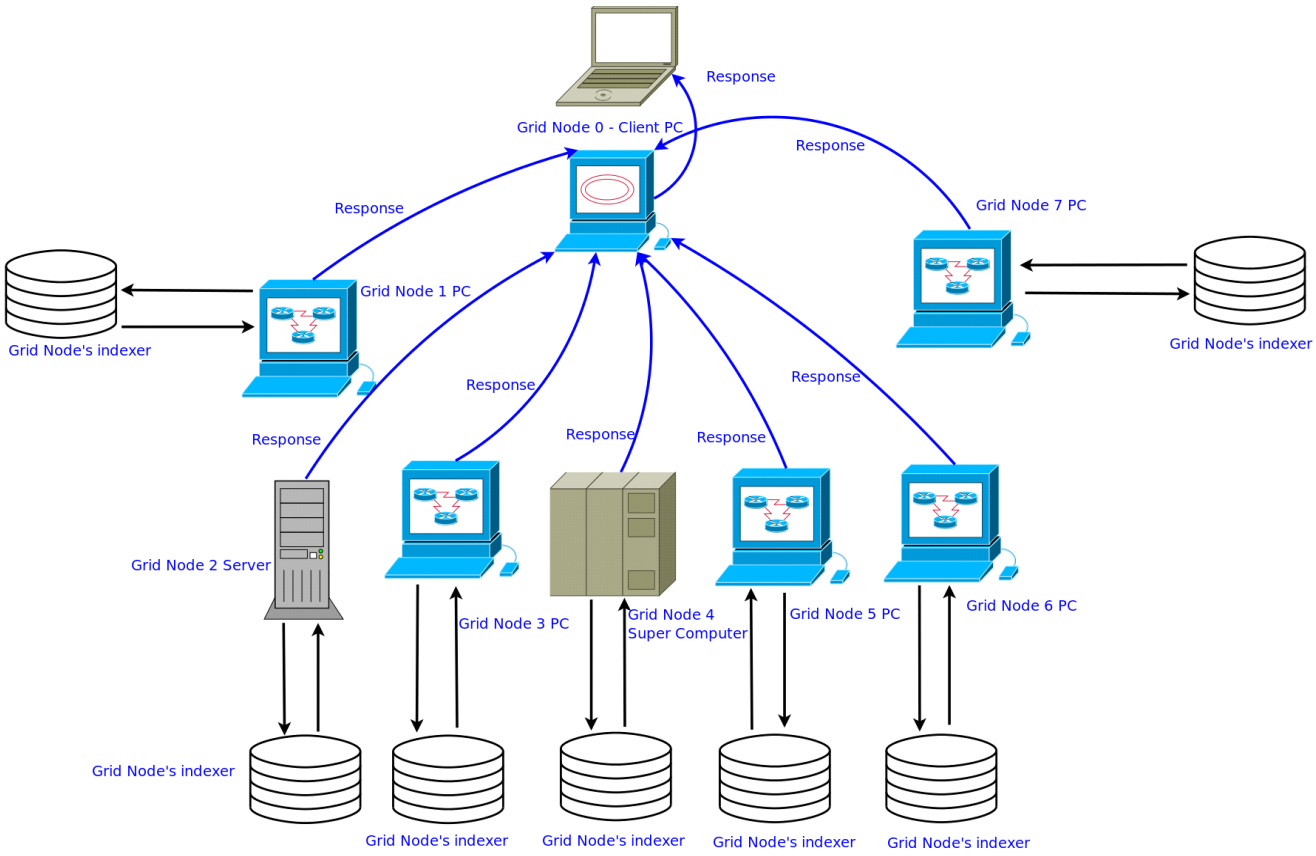


Figure 27 Multi requests and multi responses 2

Each node after receiving request process it and sends back to the client response. On the client side responses are collected from all nodes and surrounded with additional use full class, after that search procedure is finished.

Disadvantages of this approach:

1. Perhaps poor performance if there is too large list, compared with the architecture of the central indexer
2. Advantages :
3. Full asynchronicity.
4. Any node can use search procedure.
5. Each node could have different indexation system.
6. Easy to implement.
7. Doesn't need special servers or software to install.
8. Doesn't depend on grid node's chain, where whole process may fail cause of one node.

This architecture was chosen for the implementation of federated search in UNICORE. Bacause of advantages and easy to implement and it satisfy the condition of the problem.

In the “General actions sequence” section it was described via “UML Use Case” approximately model of interaction between client and system. Let’s describe model taking into account requirements which were described in “Basic requirements” section and chosen architecture:

Client want to find some data through the whole grid (for client whole grid is on virtual machine), to do this he/she defines search query – in which describes criteria of data should be found. Also client can define in which medias this data should be located – grid nodes where search should be executed.

Although as it was said federated search may take long time, therefore client at any moment should be able to know about search state. – is it finished or not, how many nodes are processed, which nodes already executed search and returned results.

It is also possible that client may leave system and search haven't been finished, it means that in the next entering the system client should be able to get results of last search he/she conducted.

It looks like this:

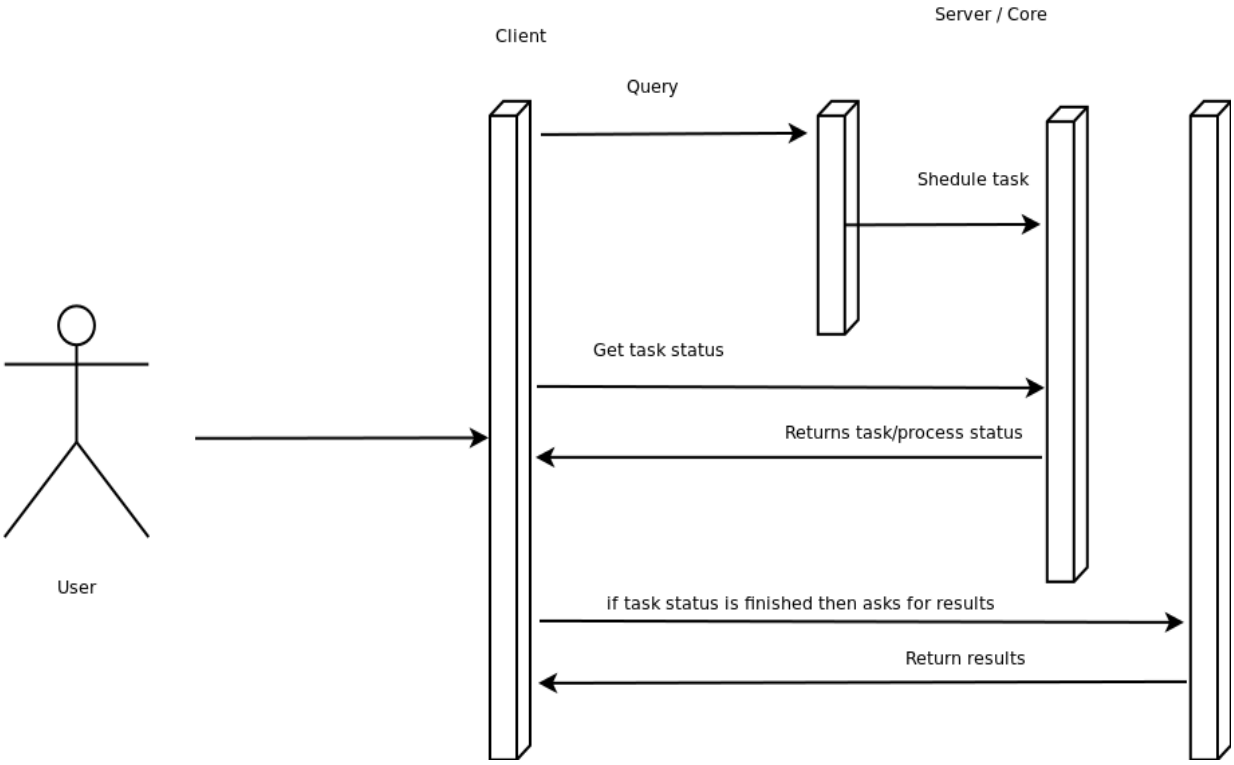


Figure 28 User and system interaction in case of federated search

Diagram above describes client server interaction:

1. Client conducts search.
2. System starts search on the defined nodes.
3. Client asks system about search statement.
4. System responses that search haven't finished and returns existing state.
5. System finishes search and returns results to the client.

Scenario above describes the probable model of the client system interaction after implementation tasks using “Multi requests and multi responses” architecture.

Search results 12

In this section it will be described which results actually client's side will get after executing federated search.

Here is shown code's key moments full code are available in appendix A

Because the files may be of various types - type which cannot be indexed– video , pictures , diagrams etc, system uses files metadata characteristic for search. It would be not correct to return metadata as a response for search – user doesn't interest in characteristic – he/she can use them for search - it would be more logical if user get location of files which satisfy search criteria – to access them.

Therefore each grid node which provided search should return address of resource which satisfies search criteria, and because search will be executed on several nodes at same time it also should return its URL.

The class which will be returned by grid nodes looks like:

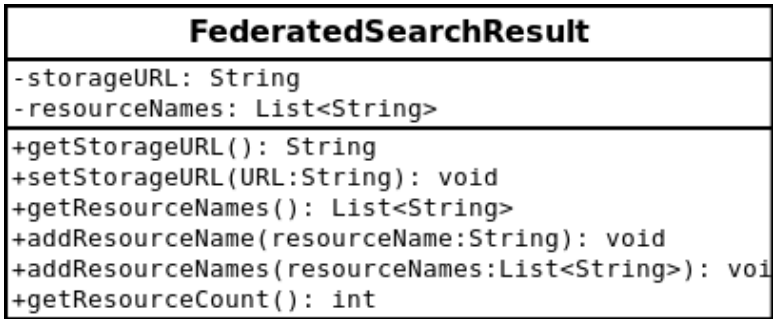


Figure 29 Class FederatedSearchResult

- storageURL – private field to store unique URL grid node.
- resourceNames – private collection field to store resources list which satisfy search criteria.

- getStorageURL() - returns value of the variable storageURL
- setStorageURL(URL : String) – sets value, derived via parameter, to the variable storageURL
- getResourceNames() – returns reosurceNames value
- addResourceName(resourceName : String) – gets resource name string and adds to the field resourceNames.
- addResourceNames(resourceNames : List<String>) - gets collection of resource names and merges with field resourceNames value.
- GetResourceCount() - returns items count in collection field resourceNames – count of found resources in the node.

Each grid node returns data in form of class described above. But client side doesn't get as a response the collection of these classes.

It was decided that results collected from all nodes will be wrapped in class with some use full information about search and data it contains – some think like meta information about search. This class is FederatedSearchResultRepository which looks like :

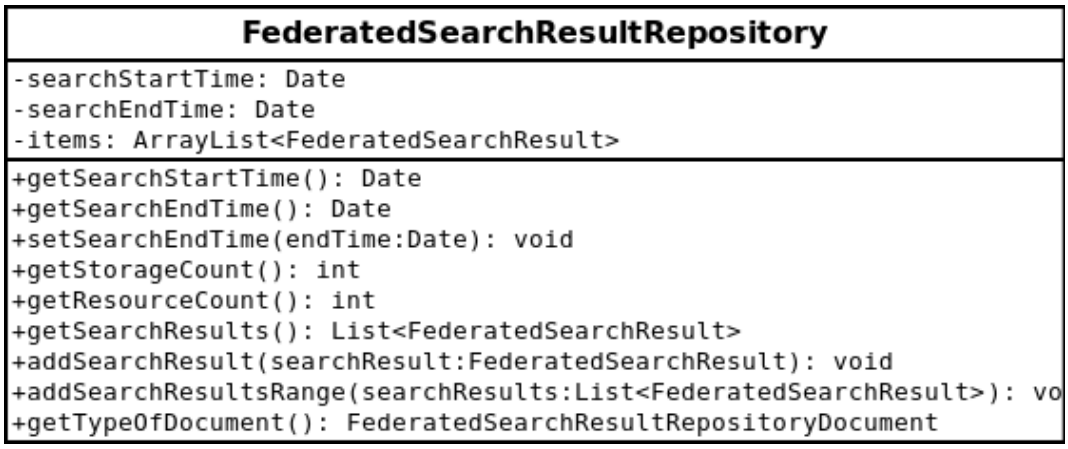


Figure 30 Class FederatedSearchResultRepository

It contains three fields:

- searchStartTime - Date type which defines when search was started .
- SearchEndTime – Date type, defines when search was finished.
- Items - ArrayList<FederatedSearchResult> type field, where are stored responses from all the nodes where search was executed.

Methods:

- getSearchStartTime() - return value of field searchStartTime – this field doesn't have setter method because it is initialized when search begins.
- SetSearchEndTime(Date endTime) method sets value to the field searchEndTime
- GetSearchEndTime() - returns value of field searchEndTime.
- GetStorageCount() - returns count of strages which responsed the search.
- GetResourceCount() - returns count of resources in all storages.
- GetSearchResults() - returns value of field items – search result.
- addSearchResult(FederatedSearchResult searchResult) – adds element to the collection field items.
- addSearchResultRange(List<FederatedSearchResult> searchResults) – adds several elements to the collection field items.
- GetTypeOfDocument() - converts class to the DocumentType – FederatedSearchResultDocument.

This class as a response gets client after execution of federated search.

Used tools and frameworks 13

This system has been written long before I started working on a federated search - not for me to decide how to implement it - by what technology or frameworks. I just want to familiar readers with frameworks and tools which were used for implementation of problem:

- Java 6 – UNICORE open source and written in Java.

- JAX-WS to create a SOAP-based web service (document style) endpoint. Because all "UNICORE"s modules interacts with each other via web services.
- Apache Maven - an innovative software project management tool provides new concept of a project object model (**POM**) file to manage project's build, dependency and documentation.
- Eclipse - as Integrated Development Tool (IDE)
- SVN – Linux subversion command line client version 1.6.17. - for source control.
- Apache Lucene - for metadata indexing.
- Apache Tika™ for detecting and extracting metadata and structured text content from various documents using existing parser libraries.
- **JUnit** as a unit testing framework. JUnit because in UNICORE it is important test – driven development.

Implementation 14

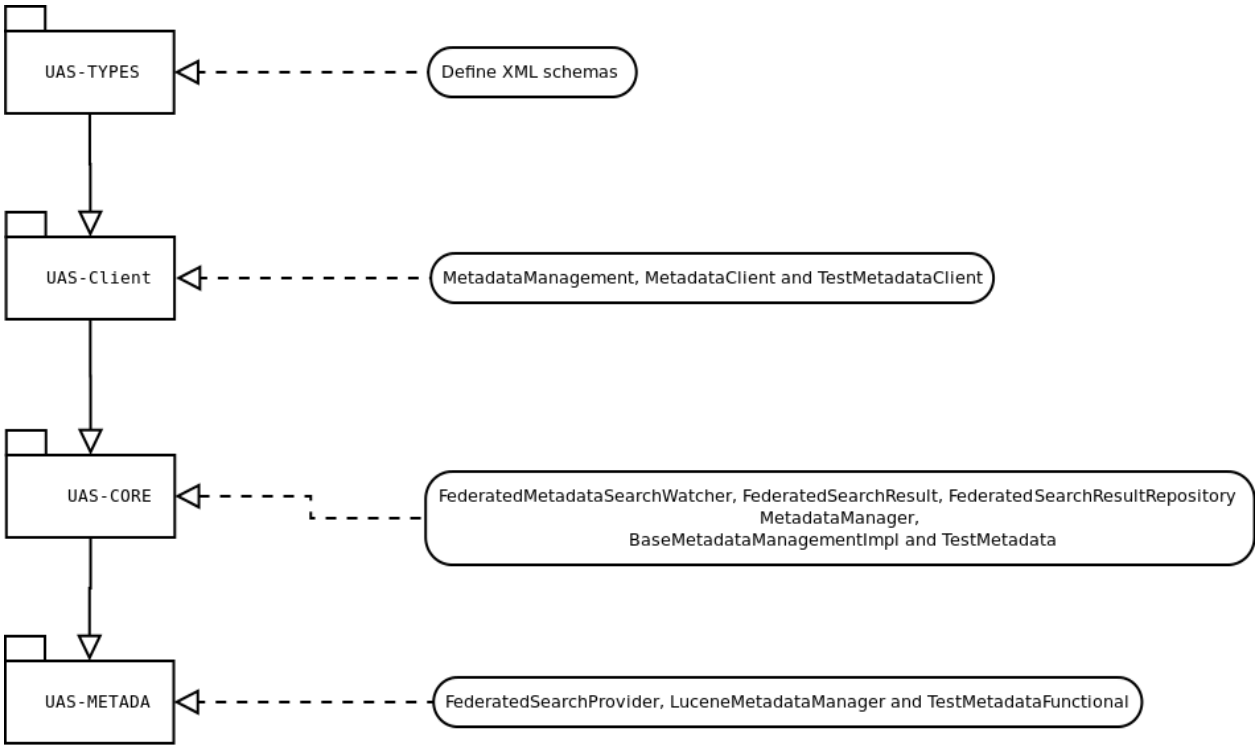


Figure 31 Development dependency chain

Based on the architecture of UNICORE implementation of "federated metadata search" had special sequence of actions/coding. Because modules are interconnected and form a logical chain first changes were made in UAS-Types module, after in UAS-Client then in UAS-Core and then in UAS-Metadata. This dependency chain and each stage developed classes / interfaces shown in diagram above.

On the first stage of development as was said changes was made in UAS-Types module. Once the signature federated metadata search's method and structure of return classes were defined (which were described in section Search results) they were described in XSD format in module UAS-Types. This XML

schema was needed in order to generate the necessary Java classes for data transferring between modules, because UNICRE has service based architecture and it uses JAX-WS.

As shown in picture below each field of that classes and method signature were described with fields and parameters types.

```
<!--  ========== Federated Metadata Search ============ -->

<xsd:element name="FederatedMetadataSearch">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="SearchString" type="xsd:string"/>
        <xsd:element name="StoragesList" type="xsd:string" minOccurs="0" maxOccurs="unbounded" />
        <xsd:element name="IsAdvanced" type="xsd:boolean"/>
        </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="FederatedMetadataSearchResponse">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="TaskReference"  type="wsa:EndpointReferenceType"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="FederatedSearchResult">
     <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="StorageURL" type="xsd:string"/>
          <xsd:element name="ResourceName" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="FederatedSearchResultRepository">
   <xsd:complexType>
     <xsd:sequence>
       <xsd:element name="SearchStartTime" type="xsd:date"/>
       <xsd:element name="SearchEndTime" type="xsd:date"/>
       <xsd:element name="StorageCount" type="xsd:integer"/>
       <xsd:element name="ResourceCount" type="xsd:integer"/>
      <xsd:element  name="FederatedSearchResults" minOccurs="0"
                                          maxOccurs="unbounded">
          <xsd:complexType>
```

Figure 32 Federated metadata search attributes description in XSD

After necessary classes were generated, it became possible to start implementation of the second stage.

On the second stage as shown in diagram above changes were made in UAS-Client module – to be precise in MetadataManagement interface, interface which represents web service's functional front end, there was added federatedMetadataSearch method which's signature shown in picture below :

```
@SEIOperationType(OperationType.read)
@WebMethod(action = "http://unigrids.org/2006/04/services/metadata/MetadataManagement/FederatedMetadataSearchRequest")
public org.unigrids.x2006.x04.services.metadata.FederatedMetadataSearchResponseDocument FederatedMetadataSearch(
        org.unigrids.x2006.x04.services.metadata.FederatedMetadataSearchDocument in)
        throws BaseFault;
```

Figure 33 federatedMetadataSearch method in UAS-Client

Also method with same name was added in MetadatClient which return TaskClient class as a response.

In MetadataClient class federatedMetadataSearch method which got next parameters:

- String query – string type search query.

- String[] storagesList – array of strings with URLs of storages where user wants to execute search.
- Boolean isAdvanced – boolean type parameter which will be passed to "native" search method.

Converts this parameters into document type and passes them to MetadataManagement's FederatedMetadataSearch service method. Key moments are shown in picture below.

```
FederatedMetadataSearchDocument federatedMetadataSearchDocument = FederatedMetadataSearchDocument.Factory.newInstance();
federatedMetadataSearchDocument.addNewFederatedMetadataSearch().setSearchString(searchString);
federatedMetadataSearchDocument.getFederatedMetadataSearch().setStoragesListArray(storagesList);
federatedMetadataSearchDocument.getFederatedMetadataSearch().setIsAdvanced(isAdvanced);

FederatedMetadataSearchResponseDocument response = metadataManagement.FederatedMetadataSearch(federatedMetadataSearchDocument);
EndpointReferenceType epr=response.getFederatedMetadataSearchResponse().getTaskReference();
TaskClient task = new TaskClient(epr, getSecurityConfiguration());
```

Figure 34 MetadataClient's federatedMetadataSearch code fragment

After these changes were done, the next step was to made changes in UNICORE's core module – UAS-Core. The first step was to add method in BaseMetadataManagementImpl class. This class implements MetadataManagement service interface, which is in UAS-Client module.

Because it was added method federatedMetadataSearch in MetadataManagement interface, it was necessary to ovveride it in BaseMetadataManagementImpl class. The overridden method gets parameter in document style. In the method we extract that parameters in simple Java type format, then create a MetadaManager inherit object instance, call its federated search method and pass parameters to it. MetadaManager which were maintioned is an interface of UAS-Core module, which describes base functionality of metadata. Because BaseMetadataManagementImpl uses this interface as tool to manipulate metadata, method signature federatedMetadaSearch was added in this interface also.

BaseMetadataManagementImpl uses MetadaManager to handle metadata because there could be various metadata implementations, not only Lucene type. On the diagram below you can see dependencies between this classes and interfaces, because it may be confusing – some classes and interfaces hase similar names in different modules.

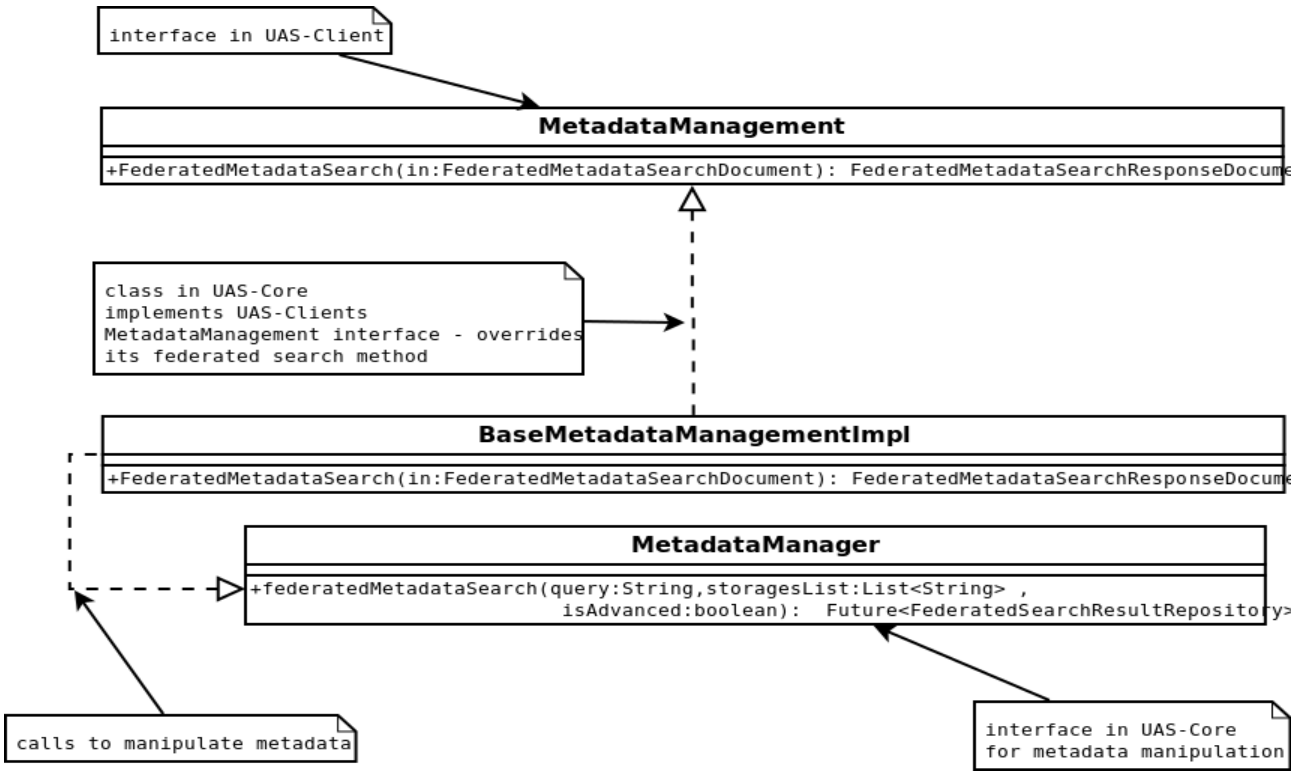


Figure 35 Class dependency

After that it came time to implement search results return type – which were described in "Search results" section, here is nothing special - everything was implemented as was described.

The next stage was – to write class FederatedMetadataSearchWatcher which implements Runnable interface. This class was needed to create a task in system.

After job was done it came time to test code written. So in TestMetadata it was added method for JUnit testing to check functionality. But what can be tested when functionality is no written? - The answer is base service functionality – send request , core module should handle it then response something. To response something there is "fake" class which implements MetadataManager interface and returns simulated expected data. So FederatedMetadataSearch was overridden and was returning fake response. Not only response was interesting for us, it was also interesting process of getting task status when method is processing and hadn't finished work. To do it, when method was called it was creating fake Future type collection of FederatedSearchRepository and calling Thread.sleep(10 000); procedure – pausing its work for 10 seconds. During that 10 second it was responding that method is processing, after that it was adding fake storages and fake resources to FederatedSearchRepository and returning response.

In TestMetadata test method was checking every second status of process for 120 seconds. If after two minutes it didn't get positive status - test considered as failed. Otherwise when it got the response it started to check all fields of FederatedSearchRepository if one of them was empty test considered as failed, else it was successful.

After service functionality was written it came time to write federated search functionality. Main changes were made in UAS-Metadata module.

Class LuceneMetadataManager - implements MetadataManager of UAS-Core. LuceneMetadataManager is designed to manipulate and index metadata via Apache Lucene. If you want to manage metadata other way – for expample via some RDBMS let's say Oracle - to store and manipulate meta then you would have to create some OracleMetaManager which would implement UAS-Core's MetadataManager and implement functionality there.

As it was said this class implements MetadataManager, what means all the methods should be overriden. In LuceneMetadataManager federatedMetadataSearch shedules service executor service to get federated search be done. To do it, method creates threading services and executor services pass the parameter and schedules service. Key moments are shown on picture below:

```
FederatedSearchProvider searchProvider = new FederatedSearchProvider(kernel, searchString, storagesList, isAdvanced);

ContainerProperties containerProperties =  kernel.getContainerProperties();
ThreadingServices threadServicies = containerProperties.getThreadingServices();
ScheduledExecutorService executorService = threadServicies.getScheduledExecutorService();
Future<FederatedSearchResultRepository> result = executorService.schedule(searchProvider, 1, TimeUnit.SECONDS);
```

Figure 36 LuceneMetadataManager federatedMetadataSearch code fragment

So to schedule service we have to pass as a parameter to executor service some class which do federated search and this class should implement Callable interface.

FederatedSearchProvider – implemets Callable interface. Main class where federated search logic is implemented in call () method.

Diagram below describes FederatedSearchProvider class.

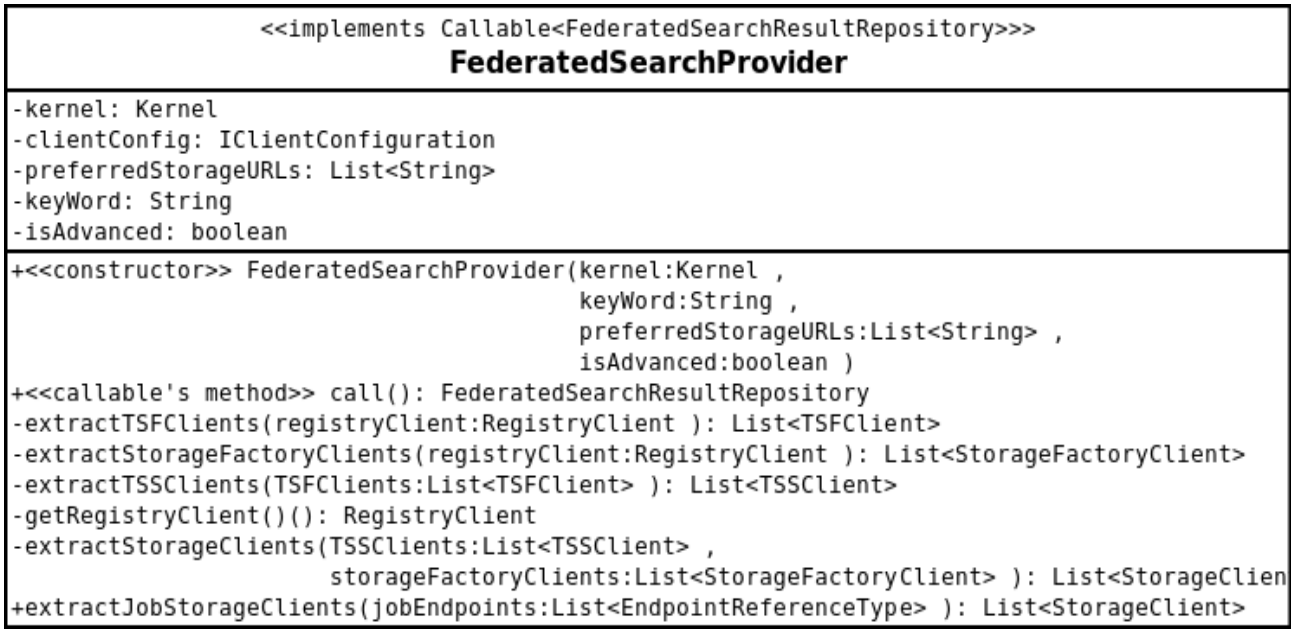


Figure 37 Class FederatedSearchProvider

- kernel – Kernel type field responsible for node's setting and getting configuration information

- clientConfig – IClientConfiguration type field responsible for configuration stuff, it's value is extracted from kernel.
- preferredStorageURLs - Collection of String type field which contains URL of nodes/sites defiened by user where search should be provided.
- isAdvanced – boolean type field which should be passed to "native" search method of grid node.
- Constructor – which gets params and initializes class fields?
- call() - method returns FederatedSearchResultRepository – comes from Callable interface, in this method is defined federated search logic.
- getRegistryClient() - method returns RegistryClient type – which is necessary for getting avaliable storages/sites.
- extractTSFClients – method returns list of TSFClient type – client for the "Target SystemFactory" service. Recevies RegistryClient as a parameters and extracts list of accessible services as endpoint reference type and creates object instance of TSFClient. Returns as a response collection of created TSFClients.
- extractTSSClients - method returns list of TSSClient type – client for the "Target System Service". Receives extracted before list of TSFClients. Walks through TSFClients list, extracts list of accessible target systems as endpoint reference type. After creates object instance of TSSClients and returns collection of new created TSSClient as a response.
- extractStorageFactoryClients – method returns list of StorageFactoryClient type – client for the "Storage Factory Service". Recevies RegistryClient as a parameters and extracts list of accessible services of storage factory types as endpoint reference type and creates object instance of StorageFactoryClient. Returns as a response collection of new created StorageFactoryClient.
- extractJobStorageClients - method returns list of StorageClient type, recevies as parameter list of endpoint reference types – job endpoints. Walks through the list, extracts endpoint adress, checks if this address is in prefared storages list – defined by user, if so – creates JobClient object. Then iterates throu the created job clients and gets storage client.
- extractStorageClients – method returns list of StorageClient type, recevies in parameters list of TSSClients and StorageFactoryClients. Iterates through this two collections, extracts accessible storages merges them. After that walks through the new created common endpoint reference type collection and extracts their addresses. If address is in prefared list of storages, then it creates storage client. In the first version of this class it first was creating storage clients and after that was filtering them by addresses and prefared storages list. But as test showed creating StorageClient object is pretty time and memory expensive operation. So it was decided that first collection of endpoint reference types should be checked if their locations are appropriated and only after that StorageClient's objects could be created. This method was redesigned to statisfy this conditions. After that it calls extractJobStorageClients() method gets results and merges them and returns common list of StorageClients.
- Method call() - overriden method interface Callable. Returns FederatedSearchResultRepository – result if federated search. All the logic of federated search is implemented in this method. Let's look implementation of this method in more detail. First it's creating result object – object type of FederatedSearchResultRepository. After creates RegistryClient object – via getRegistryClient() method described above registry client is required for getting different types of accessble services . Calls extractTFSClients() method passing as parameter registry client's object. TSFClient is necessary for extracting TSSClient. Then it extracts TSSClients via appropriative method. After that it starts extracting storage factory clients via also passing registry client to extractStorageFactoryClients() method. After that it calls extractStorageClients() and passing it

two params: extracted before TSSClients and storageFactoryClients. This procedure is necessary to get merged common list of StorageClients – via which it becomes possible to accses node's search engine and other functionality. After getting common list of storage clients, which represents storages defined in user's prefered list, it starts iteration through them. First in loop it creates federated search result item which will be added in repository result. Second step is getting MetadataClient object, described below, from storage client, to manipulate metadata. Then using metadata client it calls nodes native search method and pass to it query and isAdvanced field. After getting response it sets storage URL to federated search result item and adds to its collection type field all results returned by nodes search (each node may have different indexing/search engine) method. Then it adds created durring iterative step federatedSearchResult field - which represents search result for node, to "global" result field – federatedSearchRepository and starts next iteration. After walking through the whole list it defines search end time to field federatedSearchRepository and federated metadata search is done – returns response to client. Key moments described above actions are shown on picture below.

```
95      @Override
96      public FederatedSearchResultRepository call() throws Exception {
97
98          FederatedSearchResultRepository result = new FederatedSearchResultRepository();
99
100         RegistryClient registryClient = getRegistryClient();
101
102         List<TSFClient> TSFClients = extractTSFClients(registryClient);
103
104         List<TSSClient> TSSClients = extractTSSClients(TSFClients);
105
106         List<StorageFactoryClient> storageFactoryClients = extractStorageFactoryClients(registryClient);
107
108         List<StorageClient> list = extractStorageClients(TSSClients, storageFactoryClients);
109
110
111         for(StorageClient client : list)
112         {
113             FederatedSearchResult federatedSearchResult = new FederatedSearchResult();
114
115             MetadataClient metadataClient = client.getMetadataClient();
116
117             String storageURL = client.getUrl();
118             Collection<String> searchResult = metadataClient.search(keyWord, isAdvanced);
119
120             federatedSearchResult.setStorageURL(storageURL);
121             federatedSearchResult.addResourceNames(new ArrayList<String>(searchResult));
122
123             result.addSearchResult(federatedSearchResult);
124         }
125
126         result.setSearchEndTime(new Date());
```

Figure 38 FederatedSearchProvider's call method's code fragment

Class TestMetadataFunctional is designed to test functionality. It was added method to test FederatedMetadataSearchProvider. Method creates default storages, fake metadata metadata on each storage.

After that it calls federated metadata search and pass as parameters URL of new created storages and search query. Because created metadata is fake it can be easily calculated results depending on search

query. So adter receiving results it compares if results match with expectations, if it's so – tests passed, otherwise thest failed.

Test results on different grid configurations 15

To test performance of federated metadata search it was decided to conduct two types of tests :

1. Test performance with increasing storage count
2. Test performance with increasing files and metadata count.

First type test was provided with next parameters:

1. one file
2. one metadata file
3. Increasing storage from 1 to 10

Test results are shown on the diagram below :

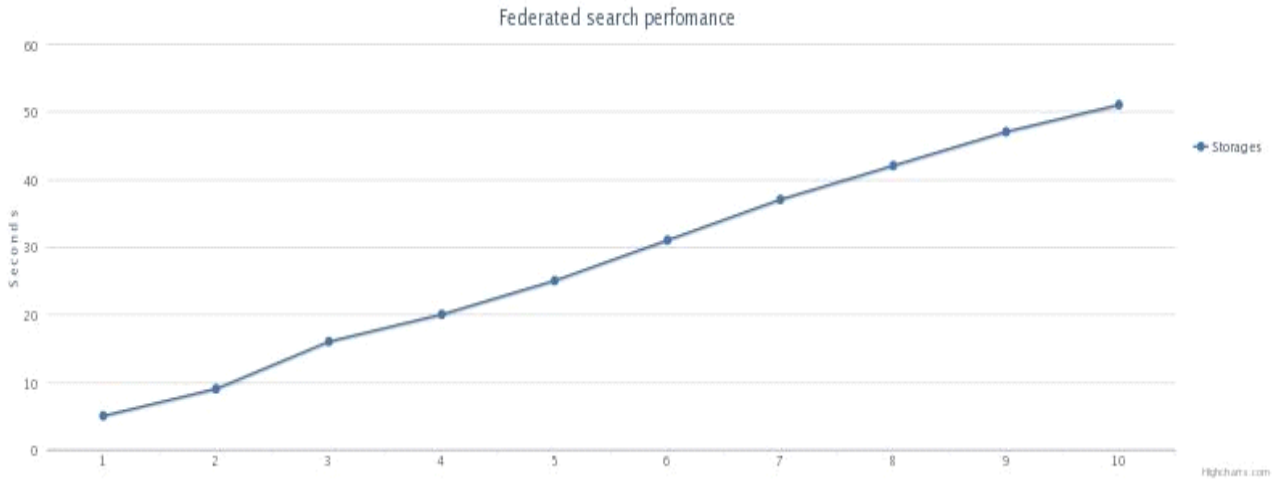


Figure 39 Federates search performance chart

With increasing of storage count processing time was increasing proportionally – if for x storage time was x with increasing storage count to 10x processing time was increased 10 times.

For the second type test there were following parameters:

1. Increasing file count from 1 to 10 and 100
2. Increasing metadata count from 1 to 10 and 100
3. One storage

Test results are shown on the diagram below:

Figure 40 Federates search performance chart 2

As shown on the diagram with metadata/file count increasing processing time almost hadn't changed – it was changing with small inaccuracy – for example if for 1 metadata file processing time was x for 100 metadata file process time still was nearly x .

From both tests we can conclude that federated metadata search is sensitive for storage count and almost insensitive for data count

Result we have and what we got 16

After this work as a result we got a convenient and necessary tool for organizing and manage data. Federated data search – is a fundamental requirement for this kind of systems/platforms – where data is distributed between several medias/storage. It was chosen stable, reliable, fast and fault tolerant architecture - multi requests multi responses. Now every user has opportunity to use federated metadata search for all or some of grid nodes, without pre indexation in real time mode.

This future is also available for UNICORE based projects like D-Grid, GRIP, OpenMolGRID, UniGrids etc, without specialized servers designed for data indexation or query distribution. Federated metadata search is not network sensitive, if client has access to the grid – then it will always be available to conduct distributed search.

So that the user would be able to use federated search it should be implemented UNICRE command-line client and / or graphic user interface via which it would be possible to provide federated search and via which user could define search query and list of nodes where search should be provided.

Also it should implemented data output for user in graphic interface part and command line client part, also should be thought over the user-friendly output format.

Conclusion 17

In this thesis described step by step implementation of federated meta data search problem in UNICORE with discussion of possible architectures and arguments why some of them were declined, tests and causes of decision making.

Implementation of this task including testing and paper writing took about five months. This implementation of federated meta data search proved its efficiency through the different tests.

It is easy to argue that federated search is useful tool for such types of platforms as UNICORE is.

Federated metadata search was successfully tested and integrated in UNICORE. Result show that federated search is almost data count independent. It will allow a high-level of flexibility of meta data management.

The next step to be done is implementing data output and may be adding features to search engine.

Appendix A 18

Here are presented key classes and methods of federated metadata search. Full version and UNICORE you can download at http://sourceforge.net/projects/unicore/ freely it is open source.

FederatedMetadataSearchWatcher

```
public class FederatedMetadataSearchWatcher implements Runnable {

    private final String taskID;

    private final Future<FederatedSearchResultCollection> future;

    private final Kernel kernel;


    public FederatedMetadataSearchWatcher(

                Future<FederatedSearchResultCollection> future, String taskID,

                Kernel kernel) {

        this.taskID = taskID;

        this.future = future;

        this.kernel = kernel;

    }


    @Override

    public void run() {

        XmlObject result = null;
```

```
                if (future.isDone()) {
                        try {
                                FederatedSearchResultCollection searchResultCollection = future.get();
                                FederatedSearchResultCollectionDocument
searchResultCollectionDocument = searchResultCollection.getTypeOfDocument();
                                result = searchResultCollectionDocument;
                                TaskImpl.putResult(kernel, taskID, result, "OK", 0);

                        } catch (Exception ex) {
                                ex.printStackTrace();
                                result = TaskImpl.getDefaultResult();
                }

                } else {
                        kernel.getContainerProperties().getThreadingServices()
                                        .getScheduledExecutorService()
                                        .schedule(this, 5000, TimeUnit.MILLISECONDS);
                }
        }
}
```

FederatedSearchResult

```
public class FederatedSearchResult implements Serializable {
        private static final long serialVersionUID = 1L;
        private String storageURL;
        private List<String> resourceNames;
        public FederatedSearchResult()
        {
                resourceNames = new ArrayList<String>();
        }
        public String getStorageURL()
        {
                return storageURL;
```

```
	}
	public void setStorageURL(String URL)
	{
		storageURL = URL;
	}

	public List<String> getResourceNames()
	{
		return resourceNames;
	}
	public void addResourceName(String resourceName)
	{
		if(resourceName == null || resourceName.isEmpty())
		{
			return;
		}
		resourceNames.add(resourceName);
	}
	public void addResourceNames(List<String> resourceNames)
	{
		this.resourceNames.addAll(resourceNames);
	}
	public int getResourceCount()
	{
		return resourceNames.size();
	}
}
```

FederatedSearchResultCollection

```
public class FederatedSearchResultCollection {
	Date searchStartTime;
	Date searchEndTime;
	ArrayList<FederatedSearchResult> items;
```

```
public FederatedSearchResultCollection() {
        searchStartTime = new Date();
        items = new ArrayList<FederatedSearchResult>();
}
public Date getSearchStartTime() {
        return searchStartTime;
}
public Date getSearchEndTime() {
        return searchEndTime;
}
public void setSearchEndTime(Date endTime) {
        if (searchStartTime.after(endTime)) {
                throw new IllegalArgumentException(
                                "The specified search end time is not valid - it's less than start
                                time.");
        }
        searchEndTime = endTime;
}
public int getStorageCount() {
        return items.size();
}
public int getResourceCount() {
        int result = 0;
        for (FederatedSearchResult item : items)
                result += item.getResourceCount();
        return result;
}
public List<FederatedSearchResult> getSearchResults() {
        return items;
}
public void addSearchResult(FederatedSearchResult searchResult)
{
        items.add(searchResult);
```

```
	}
	public void addSearchResultsRange(List<FederatedSearchResult> searchResults)
	{
		items.addAll(searchResults);
	}
	public FederatedSearchResultCollectionDocument getTypeOfDocument()
	{
		FederatedSearchResultCollectionDocument result =
							FederatedSearchResultCollectionDocument.Factory
				.newInstance();
		result.addNewFederatedSearchResultCollection();
result.getFederatedSearchResultCollection().setSearchEndTime(DateToCalendar(getSearchStartTime()));
result.getFederatedSearchResultCollection().setSearchEndTime(DateToCalendar(getSearchEndTime()));
result.getFederatedSearchResultCollection().setResourceCount(BigInteger.valueOf(getResourceCount()));
result.getFederatedSearchResultCollection().setStorageCount(BigInteger.valueOf(getStorageCount()));
		List<FederatedSearchResult> searchResults = getSearchResults();
		for(FederatedSearchResult searchResult : searchResults)
		{
	org.unigrids.x2006.x04.services.metadata.FederatedSearchResultDocument.FederatedSearchResul
t
searchResultElement=result.getFederatedSearchResultCollection().addNewFederatedSearchResults().add
NewFederatedSearchResult();
			searchResultElement.setStorageURL(searchResult.getStorageURL());
			List<String> elementResourceNames = searchResult.getResourceNames();
			String[] elementResourceNamesArray = elementResourceNames.toArray(new
String[elementResourceNames.size()]);
			searchResultElement.setResourceNameArray(elementResourceNamesArray);
		}
		return result;
	}
	public Calendar DateToCalendar(Date date){
		  Calendar cal = Calendar.getInstance();
		  cal.setTime(date);
		  return cal;
```

```
        }

}
```

FederatedMetadataSearchResponseDocument of UAS-CORE in BaseMetadataManagementImpl

```
@Override
public FederatedMetadataSearchResponseDocument FederatedMetadataSearch(
FederatedMetadataSearchDocument in) throws BaseFault {
        try {
                FederatedMetadataSearch search = in.getFederatedMetadataSearch();

                String searchString = search.getSearchString();
                boolean isAdvanced = search.getIsAdvanced();
                String[] storagesListArray = search.getStoragesListArray();
                List<String> storagesList = Arrays.asList(storagesListArray);

                MetadataManager manager = getMetadataManager();

                Future<FederatedSearchResultCollection> future = manager
                .federatedMetadataSearch(searchString, storagesList, isAdvanced);
                FederatedMetadataSearchResponseDocument response =
                FederatedMetadataSearchResponseDocument.Factory
                .newInstance();
                response.addNewFederatedMetadataSearchResponse();

                EndpointReferenceType eprt = makeFederatedSearchTask(future);
                response.getFederatedMetadataSearchResponse()
                .setTaskReference(eprt);

                return response;
        } catch (Exception ex) {
        ex.printStackTrace();
        LogUtil.logException(
        "Error occurred during federated metadata search.", ex,logger);
        throw BaseFault.createFault(
        "Error occurred during federated metadata search.", ex,
        true);
        }
}
```

FederatedSearchProvider of UAS-METADATA

```
public class FederatedSearchProvider implements
Callable<FederatedSearchResultCollection> {

private final Kernel kernel;
private final IClientConfiguration clientConfig;
private final List<String> preferredStorageURLs;
private final String keyWord;
private final boolean isAdvanced;

public FederatedSearchProvider(Kernel kernel
, String keyWord
, List<String> preferredStorageURLs
, boolean isAdvanced) {
                this.kernel = kernel;
                this.clientConfig = kernel.getClientConfiguration();
                this.keyWord = keyWord;
                this.preferredStorageURLs = preferredStorageURLs;
                this.isAdvanced = isAdvanced;
}

@Override
public FederatedSearchResultCollection call() throws Exception {

        FederatedSearchResultCollection result = new FederatedSearchResultCollection();

        RegistryClient registryClient = getRegistryClient();
        List<TSFClient> TSFClients = extractTSFClients(registryClient);
        List<TSSClient> TSSClients = extractTSSClients(TSFClients);
        List<StorageFactoryClient> storageFactoryClients =
        extractStorageFactoryClients(registryClient);
        List<StorageClient> list = extractStorageClients(TSSClients, storageFactoryClients);
        for(StorageClient client : list)
        {
                FederatedSearchResult federatedSearchResult = new FederatedSearchResult();
                MetadataClient metadataClient = client.getMetadataClient();
                String storageURL = client.getUrl();
                Collection<String> searchResult = metadataClient.search(keyWord, isAdvanced);

                federatedSearchResult.setStorageURL(storageURL);
                federatedSearchResult.addResourceNames(new ArrayList<String>(searchResult));
                result.addSearchResult(federatedSearchResult);
```

```
        }
        result.setSearchEndTime(new Date());
        return result;
}

private List<TSFClient> extractTSFClients(RegistryClient registryClient) throws Exception {
        List<TSFClient> result = new ArrayList<TSFClient>();
        List<EndpointReferenceType> TSFEndpointReferenceTypes =
        registryClient.listAccessibleServices(TargetSystemFactory.TSF_PORT);
        for (EndpointReferenceType endpoint : TSFEndpointReferenceTypes)
                result.add(new TSFClient(endpoint, clientConfig));
        return result;
        }
private List<StorageFactoryClient> extractStorageFactoryClients(RegistryClient registryClient)
throws Exception
        {
                List<StorageFactoryClient> result = new ArrayList<StorageFactoryClient>();
                List<EndpointReferenceType> SFClients =
                registryClient.listAccessibleServices(StorageFactory.SMF_PORT);
                for(EndpointReferenceType endpoint : SFClients)
                        result.add(new StorageFactoryClient(endpoint, clientConfig));
                return result;
        }

private List<TSSClient> extractTSSClients(List<TSFClient> TSFClients) throws Exception
{
        List<TSSClient> result = new ArrayList<TSSClient>();
        List<EndpointReferenceType> TSSEndpointRefenceTypes = new
        ArrayList<EndpointReferenceType>();
        for(TSFClient tsf : TSFClients){
                List<EndpointReferenceType> items = tsf.getAccessibleTargetSystems();
                if(items == null || items.size() == 0)
                        continue;
                TSSEndpointRefenceTypes.addAll(items);
        }
        for(EndpointReferenceType endpoint : TSSEndpointRefenceTypes)
                result.add(new TSSClient(endpoint, clientConfig));
        return result;
        }
private RegistryClient getRegistryClient() throws Exception {
        String url =
kernel.getContainerProperties().getValue(ContainerProperties.WSRF_BASEURL);
```

```
        EndpointReferenceType endpointReferenceType =
        EndpointReferenceType.Factory.newInstance();
        endpointReferenceType.addNewAddress().setStringValue(url + "/" +
        Registry.REGISTRY_SERVICE + "?res=default_registry");
        RegistryClient registryClient = new RegistryClient(endpointReferenceType,
        kernel.getClientConfiguration());

        return registryClient;
        }

private List<StorageClient> extractStorageClients(List<TSSClient> TSSClients,
List<StorageFactoryClient> storageFactoryClients) throws Exception
{
        List<StorageClient> result = new ArrayList<StorageClient>();
        List<EndpointReferenceType> jobEndpoints = new
        ArrayList<EndpointReferenceType>();
        List<EndpointReferenceType> storageEndpoints = new
        ArrayList<EndpointReferenceType>();
        for(TSSClient client : TSSClients)
        {
                jobEndpoints.addAll(client.getJobs());
                storageEndpoints.addAll(client.getStorages());
        }
        for(StorageFactoryClient client : storageFactoryClients)
                storageEndpoints.addAll(client.getAccessibleStorages());
        for(EndpointReferenceType endpoint : storageEndpoints)
        {
                String endpointAddress = endpoint.getAddress().getStringValue();
                if(preferredStorageURLs.contains(endpointAddress)) // creates only if and point
                address is in preferred storages list
                result.add(new StorageClient(endpoint, clientConfig));
        }
        result.addAll(extractJobStorageClients(jobEndpoints));
        return result;
}

private List<StorageClient> extractJobStorageClients(List<EndpointReferenceType>
jobEndpoints) throws Exception
{
        List<StorageClient> result = new ArrayList<StorageClient>();
        List<JobClient> jobClients = new ArrayList<JobClient>();
        for(EndpointReferenceType endpoint : jobEndpoints)
        {
```

```
                String endpointAddress = endpoint.getAddress().getStringValue();
                if(preferredStorageURLs.contains(endpointAddress))
                jobClients.add(new JobClient(endpoint, clientConfig));
        }
        for(JobClient client : jobClients)
        result.add(client.getUspaceClient());
        return result;
        }
}
```

References 19

[1] M. Velicanu, I. Lungu, I. Botha, A. Bara, A. Velicanu, E. Rednic – Sisteme de baze de date evaluate, Ed. Ase, Bucuresti 2009

[2] A. Abbas - Grid Computing: A Practical Guide to Technology and Applications, Editura Charles River Media, 2005

[3] Grid Computing Technology - Georgiana MARIN Romanian – American University Bucharest, ROMANIA

[4] Grid computing (https://en.wikipedia.org/wiki/Grid_computing)

[5] UNICORE official web site (http://www.unicore.eu/unicore/architecture.php)

[6] UNICORE (http://en.wikipedia.org/wiki/UNICORE)

[7] Joint Project Reportfor the BMBF ProjectUNICORE (http://www.unicore.eu/documentation/files/erwin-2003-UPF.pdf)

[8] UNICORE Architecture (http://www.unicore.eu/unicore/architecture.php)

[9] UNICORE Summit 2012

[10] UNICORE Summit 2011

[11] Metadata (https://en.wikipedia.org/wiki/Metadata)

[12] MCAT

[13] GLite AMGA (http://en.wikipedia.org/wiki/GLite-AMGA)

[14] Replication in AMGA (http://amga.web.cern.ch/amga/replication.html)

[15] Support for MetadataFederation in AMGA
(http://indico.egi.eu/indico/getFile.py/access?contribId=105&sessionId=16&resId=0&materialId=slides&confId=207)

Images References 20

[1] - http://nabis1.at.ua/_pu/3/32658792.jpg Search logo

[2] - https://www.seegrid.csiro.au/wiki/pub/Compsrvices/DataResults/APAC-DataSearch-UseCaseDiagram1.gif User data and metadata management

[3] - http://blog.octo.com/wp-content/uploads/2010/09/TaskDistribution1.png Grid infrastructure

[4] - http://www.cloudbus.org/intergrid/images/intergrid_layers.jpg Inter grid

[5] - http://full-text.com/fig/9/2.png Grid with different nodes

[6] - http://1.bp.blogspot.com/-K7vcj8w_MLU/TaigxngU-DI/AAAAAAAADrU/GnOOycP8t-I/s1600/Screenshot-1.png Distributed systems and grid

[7] - http://sourceforge.net/p/unicore/screenshot/173301.jpg UNICORE logo

[8] - http://www.unicore.eu/unicore/architecture/architecture_no-standards_531x472.png UNICORE architecture

[9] - http://www.unicore.eu/unicore/architecture/ucc.png UNICORE Command line client

[10] - http://www.unicore.eu/unicore/architecture/client_screenshot_gamess.png UNICORE Rich client

[11] - http://www.w3.org/wiki/images/7/73/OBI_Definition_Source$metadata_properties.png Metadata

[12] - http://www.sdsc.edu/srb/images/0/0f/Mcatimg001.gif MCAT Architecture

[13] - http://www.sdsc.edu/srb/images/9/98/Mcatimg002.gif MCAT Architecture

[14] [15] [16] - AMGA presentation
http://indico.egi.eu/indico/getFile.py/access?contribId=105&sessionId=16&resId=0&materialId=slides&confId=207