# A Metadata Service for an Infrastructure of Large Scale Distributed Scientific Datasets


Oluwalani A. Adeleke

Faculty Of Science

University Of The Witwatersrand, Johannesburg

A Dissertation submitted to the Faculty Of Science in fulfilment of the requirements for the degree of

*Master Of Science by Research only*

signed on 17th February, 2014 in Johannesburg

## Declaration

I declare that this Dissertation is my own, unaided work under the supervision of Dr Dmitry Shkatov. It is being submitted for the Degree of Master of Science at the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination at any other University.

_____

_____day of _____20 _____in _____

List of Publications

1. **Conference Paper**: An Integrated Metadata Access Interface for a Network of Federated Curated Data Repositories.
   Presented at ADSC5 Conference, Durban, South Africa, June 2013.
2. **Journal Paper**:An Improved Metadata Access Interface for a Network of Federated Curated Data Repositories.
   OCLC Systems & Services:International digital library perspectives Journal 2013 (Under Peer Review).
3. **M&D Symposium**: A Metadata Service for an Infrastructure of Large Scale Distributed Scientific Datasets.
   SAICSIT Conference 2012, Centurion South Africa, Oct 2012.

## 0.1 List of Abbrevations

- **VUMC:** Virtual Unified Metadata Catalog
- **THREDDS:** (Thematic Realtime Environmental Distributed Data Services)
- **iRODS:** Integrated Rule-Oriented Data System
- **MCS:** Metadata Catalog Service
- **AVU:** Attribute, Value, Unit - The elements used for storing metadata records

# Abstract

In this constantly growing information technology driven era, data migration and replication pose a serious bottleneck in the distributed database infrastructure environment. For large heterogeneous environments with domains such as geospatial science and high energy physics, where large array of scientific data are involved, diverse challenges are encountered with respect to dataset identification, location services, and efficient retrieval of information. These challenges include locating data sources, identifying effective transfer route, and replication, just to mention a few. As distributed systems aimed at constant delivery of data to the point of query origination continue to expand in size and functionality, efficient replication and data retrieval systems have subsequently become increasingly important and relevant. One such system is an infrastructure for large scale distributed scientific data management. Several data management systems have been developed to help manage these fast growing datasets and their metadata. However little work has been done on allowing cross-communication and data-sharing between these different dataset management systems in a distributed, heterogeneous environment.

This dissertation addresses this problem, focusing particularly on metadata and provenance service associated with it. We present the Virtual Unified Metadata architecture to establish communication between remote sites within a distributed heterogeneous environment using a client-server model. The system provides a framework that allows heterogeneous metadata services communicate and share metadata and datasets through the implementation of a communication interface. It allows for metadata discovery and dataset identification by enabling remote query between heterogeneous metadata repositories. The significant contributions of this system include:

- the design and implementation of a client/server based remote metadata query system for scientific datasets within distributed heterogeneous dataset repositories;
- Implementation of a caching mechanism for optimizing the system performance;

– Analyzing the quality of service with respect to correct dataset identification, estimation of migration and replication time frame, and cache performance.

To ...

## Acknowledgements

I would like to acknowledge God Almighty for the gift of life and grace to accomplish this work. I will also like to thank my supervisor, Dr Dmitry Shkatov, for his support during this process. Special thanks to my parents Deacon and Mrs S.O Adeleke for their great financial and moral support. The people with whom I've shared an office (D. Osikoya, G. Nimako, and D. Ohene-Koffi) have been great company and I'll like to thank them for their friendship, moral support, and willingness to provide ideas where required.

Thank you all.

# Contents

# List of Figures

# Chapter 1

# Introduction

Distributed computing systems provide an underlying Information Technology (IT) infrastructure for coordinating a collection of homogeneous and heterogeneous systems that work together to achieve specific tasks over a network. They consist of several protocols, applications and services that are fused together to create a virtual system aimed at achieving resource sharing, concurrency, scalability, fault tolerance and transparency, and are largely used nowadays to solve large computational problems. Some common examples of the success of distributed computing include the increased usage of data grid systems such as the BOINC, an open-source software for volunteer computing and grid computing (berkeley.edu 2012), Milkway@Home used largely in astronomy, astro-informatics and computer science (milkyway.cs.rpi.edu 2012), and Globus, an open source Grid software that addresses the most challenging problems in distributed resource sharing (globus.org 2012). With the increased dependency on computing systems for coordinating information management, reliance on distributed computing systems has also become common place in scientific research environments. They are largely employed for achieving effective management of scientific resources from the data creation stage to storage and also for coordinating, the distribution of these scientific resources. However with the use of these large distributed computing systems, scientific research in different disciplines carrying out complex computations increasingly generate and assemble large quantities of data. These datasets generated are a combination of previous research results and new hypothesis carried out by scientists, and data generated are eventually used for further research or to provide immediate results to diverse hypothesis and theories. These large datasets derived as results from scientific research work are particularly prevalent in scientific domains such as astronomy, geosciences, paleosciences, and high energy physics. It is interesting to know that nowadays, these

datasets span up to several terabytes and sometimes run into petabytes of data (Chervenak et al. 1999). Research analysis done by the Southampton Regional e-Science Center (Johnston et al. 2008) has shown that data storage density in scientific databases doubles every 12 - 18 months. This rapid exponential growth of scientific datasets due to advances in scientific experiments, computations and research has subsequently increased the challenges involved with managing dataset repositories. It is inevitable that access to these datasets is required by a large community of researchers who most often than not are spread over dispersed geographical locations. This calls for techniques and technologies that allow scientific researchers share their datasets and repositories. Access and analysis of these datasets which are also stored and managed over dispersed network domains, is often carried out remotely using sophisticated infrastructures and management techniques.

Scientific datasets are often organized and stored using distributed data repositories equipped with distributed database management systems for organizing and managing them in storage area networks. In larger heterogeneous environments, data grid architectures are used to organize large amounts of geographically distributed data. Repository management systems therefore need to be put in place to manage these datasets in storage, and also manage the process of organizing them within databases. Some considerations must be put in place when determining the techniques and infrastructures for creating and accessing these datasets within a repository system. For instance, in ensuring ease of datasets retrievals, a good location service must be available, also considerations must be given to the appropriate techniques for obtaining the storage location of the datasets within the distributed database environment using metadata. We must also consider mechanism for extracting the datasets using the metadata information, finding effective route of access to the datasets (shortest path, or most secure path), retrieving the required datasets from source to destination (Allcock et al. 2005).

Institutions that maintain large array of electronic records or data in large storage data centers, which could be either in a single geographical location or in dispersed heterogeneous locations, therefore require a good repository system to manage and allow access to their stored data. These repository systems are usually employed for managing data archives and storage systems.

They support functional capabilities that enable a wide range of activities from data storage management, data access, metadata creation and retrievals, replica management amongst others, which are achieved using a wide range of tools and application programming interfaces (API's), that coordinate the different activities and operations within the data grids (Chervenak et al. 1999).

A good distributed repository management infrastructure should be scalable and reliable with good propensity for expansion and growth which can accommodate significant constant increase in dataset sizes over time and with constant usage. It must also allow for ease of resource sharing amongst nodes within the system (Allcock et al. 2005). The ability to use these repository management systems to organize and coordinate data repositories efficiently, promotes ease of data management (i.e data storage, access and metadata management) and also simplifies data sharing.

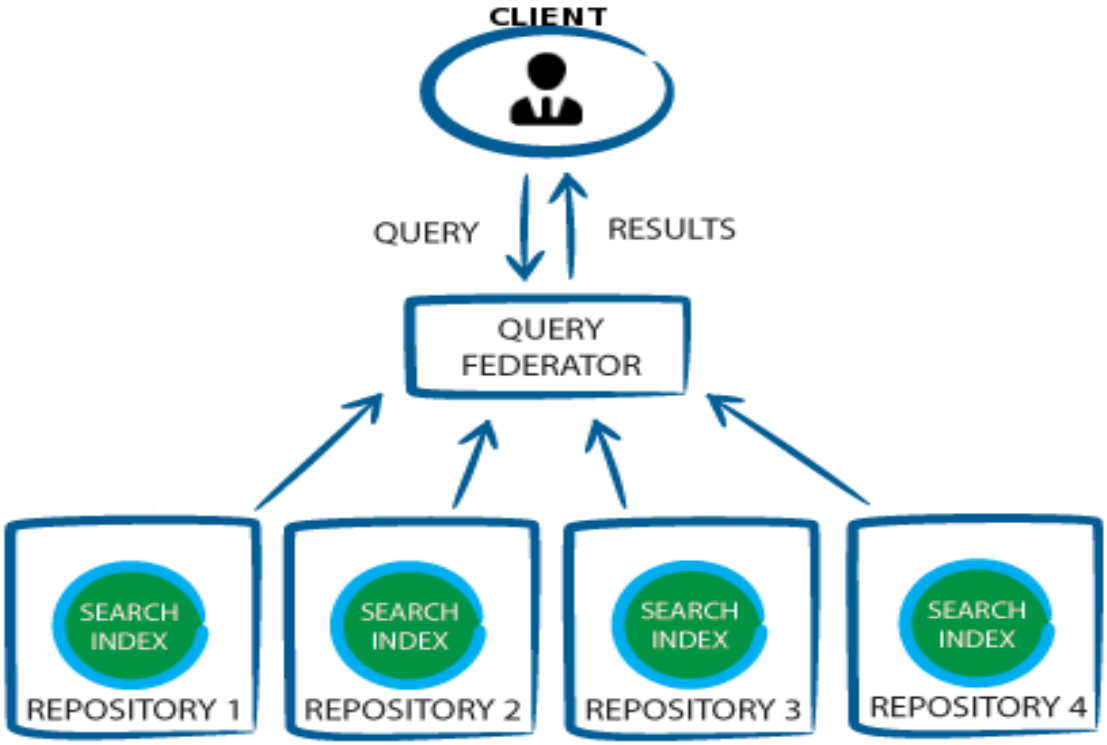


**Figure 1.1:** Federated Data Repository
(Fletcher 2013)

Federated data repositories 1.1 provide a model for structured datasets storage and sharing among heterogeneous sources and users. The notion of federalism has increased the adoption of geographically distributed data repositories which comprise different data centers under different administrative domains. These data centers contain multiple independent servers running several composite services that need to be integrated for computational purposes and problem solving. With total control of these systems being assigned to repository owners, the data management techniques largely vary within individual peers. Real time acquisition of datasets is required to ensure seamless operations between peers that require resources from neighboring peers to perform other operations. For this to be easily achieved, it is important to have architecture in place that allows ease of data discovery from all participating nodes in the federated data repositories.//

Metadata plays an important role in ensuring discovery of datasets by providing relevant information for locating these datasets. By keeping detailed metadata information, large sets of data stored within distributed data repositories can posses distinct identities which helps to reduce the complexity of locating them in a pool of thousands and sometimes millions of datasets records. A Metadata Service creates a standardized platform for sharing metadata by providing the needed tools for easy dataset discovery from federated curated data repositories. It allows for metadata sharing among the various data repositories (Liu et al. 2007). Many existing repository management systems contain reliable metadata services for managing its metadata information. These metadata services often conform to generic metadata standards such as the Dublin core metadata standard (Apps 2005), Darwin core metadata standard (Tim Robertson et al. n.d.), or specialized custom made metadata standards, to provide rich search-able metadata content for ease of data discovery. Some examples of such metadata services used largely within scientific environments include systems such as the Integrated Rule-Oriented Data System(iRODS) (DICE 2009), a data grid software system developed by the Data Intensive Cyber Environments (DICE) research group for managing digital collections, OpenDAP Hyrax a high performance DAP-compliant data server specifically for earth system grid project with existing software developed by OPeNDAP (Cornillon et al. 2003), THREDDS (Thematic Realtime Environmental Distributed Data Services) (UNIDATA n.d.) a web server that provides metadata

and data access for scientific datasets, using OPeNDAP, OGC WMS and WCS, HTTP, and other remote data access protocols developed at unidata program center, University Corporation for Atmospheric Research (UCAR), Metadata Catalog service (MCS) and Dspace, an open source repository software package developed at MIT primarily for creating open access repositories for scholarly documents and published digital content (Tim et al. n.d.) etc.

## 1.1 Problem Motivation

Metadata services (such as iRODS, OpenDAP Hyrax, MCS, Thredds etc) have been implemented for simplifying the process of organizing datasets in distributed database environments. They manage rich metadata content that allow datasets to be discovered and retrieved easily. They are organized mainly in client-server architectures where the server carries out most of the computational capabilities while the client mainly provides an interface for users to request operations to be carried out. Since system owners and administrators have total control over management of these data repositories the choice of metadata service is also their prerogative. In a distributed heterogeneous data repositories environment it is therefore conceivable that the metadata services contained in the participating data repositories will also be heterogenoeus and vary from one repository to another.

A problem prevalent with this setup however is that these repository systems do not have the ability to interoperate amongst themselves. For instance an iRODS client cannot make queries directly to a Thredds server and vice versa. In the same way, a server running in the MCS environment cannot be queried directly for metadata by an openDAP Hyrax client. Metadata sharing is only limited to environments that provide or utilize homogeneous repository systems and metadata services and this restricts the usage of such datasets. This posses a problem when institutions with different repository management systems need to collaborate and share resources managed with varying existing systems. The main research question that arises here is:

***"How can we make different repository systems communicate within a distributed dataset environment and appear like a non-distributed system?"***

This forms the basic focus of this study and we will be looking at best practices in tackling this problem and hope to provide a solution that will largely suit existing architectures.

## 1.2 Contributions

For effective information sharing to be achieved among these heterogeneous metadata services, an infrastructure needs to be put in place to enable communication and dataset sharing amongst them. Due to the differences in architecture of these various metadata services, metadata integration is not always feasible between different autonomous metadata services, and there is no communication across the different platforms.

The major area of contribution in this thesis is the development of an infrastructure that allows for metadata retrieval within these different metadata services in an heterogeneous distributed data repositories. Our framework provides a platform for interaction between varying repository management systems used in maintaining these distributed data repositories and allows for remote communication and metadata sharing by utilizing available API's for the different repository management system's.

The specific contributions areas we will outline within this thesis are:

- Development and implementation of the "Virtual Unified Metadata Catalog" (VUMC) a client-server system which provides a framework for metadata service integration, thus establishing remote metadata sharing amongst distributed dataset repositories
- Integration of some existing metadata services within the VUMC to query and retrieve metadata and detailing algorithms for achieving this
- Analyzing and estimating performance of the VUMC with respect to accuracy of dataset retrieval within the distributed environment, and measuring efficiency and reliability of

the VUMC system.

## 1.3 Organisation Of the Thesis

The remainder of this thesis is organised as follows. Chapter 2 gives a brief survey of current trends in distributed dataset management, metadata services employed and metadata standards. In Chapter 3, we describe the VUMC with its basic properties and features. The structure is described with discussions on the basic functional components. We introduce Zero C ICE, which provides the main communication interface for our VUMC system. We also describe an integrated cache design implemented within the system. Next we do an analytical survey of the VUMC model in Chapter 4 and also discuss analytical measures for its caching interface. Chapter 5 provides the detailed implementation of the VUMC by integrating three existing metadata services. We go further to break down the specific algorithms for integrating these metadata services and also for the caching infrastructure. In Chapter 6, we discuss some experiments performed, with details of the experimental environment and setup. We also provide some performance graphs for the experiments condicted. We finally summarize the work of this research in Chapter 7, and give some directions for future work.

# Chapter 2

# Literature Review and Related Work

Management of large scale distributed scientific datasets require the use of specialized architectures, techniques and tools. These dataset collections which are usually stored as multidimensional arrays require unique technology for easy storage, retrieval, managing, and analyzing. Some common formats for representing these large array of datasets include the NetCDF, HDF5 (Dursi 2012). However there are various challenges experienced in organizing and maintaining these datasets in collections. In most cases data providers are not the eventual data consumers, and therefore need to make the datasets available to their data consumers. One challenge faced as a result is in ensuring fast and efficient access to these datasets by eventual users. Also high performance computing techinques for evaluation of this datasets within an heterogeneous environment needs to be put under consideration, just to mention a few. Much work has been achieved in the area of storage and retrieval of these datasets. As earlier mentioned in the previous chapter, metadata services reduce challenges involved in dataset management and discovery in large scale distributed dataset repositories. Some relevant systems in metadata management for large scale scientific datasets include Integrated Rule-Oriented Data System (iRODS) (DICE 2009), OpeNDAP Hyrax, (Cornillon et al. 2003), Thematic Real-time Environmental Distributed Data Services (THREDDS) (UNIDATA n.d.) etc. They provide a framework for archival and retrieval of metadata information, and employ various metadata standards for metadata processing. Some of the functions of these metadata services include metadata creation, metadata query processing, dataset management, and metadata publishing (Deelman et al. 2004). Other such systems include the RasDaMan DBMS (Reiner & Hahn 2004) which optimizes management of large-scale data sets for storage and retrieval on tertiary storage systems. It achieves this using a Super-Tile algorithm and techniques like inter

and intra Super-Tile clustering, caching and scheduling, for reducing tertiary storage access times. Another such system is the Active Data Repository (ADR), which focuses on storage, retrieval, and processing of large multi-dimensional scientific datasets on distributed memory parallel machines with multiple disks attached to each node (Chang et al. 2000).

Locating stored datasets is critical if the datasets are to be retrieved for subsequent usage. In locating large-scale distributed scientific datasets, the metadata service simplifies the process of identifying these datasets and making it discoverable within the distributed environment. The metadata service is employed for organizing these digital datasets in storage and ensuring ease of location of required information from these datasets. Within this chapter we will like to look at some key features of the metadata service and also discuss some relevant systems that process metadata for large-scale distributed scientific datasets.

## 2.1 Metadata

In scientific domains, researchers depend on the existence of reliable, consistent metadata either as a means to define information for identification of their datasets by internal resources (i.e. supporting producers in locating and using their own data resources) or as a means to reliably exchange information between and among data consumers. By providing information about the contents of a data repository, it enables users to obtain quick accesses to clear, accurate, and reliable information. You are likely to query a metadata service when searching for data or information because of its ease of data retrieval. Also it is better to publish your data to a metadata service if you want it to be easily accessible to potential users. In addition to resource discovery, metadata helps to organize datasets in digital repositories, facilitate interoperability and legacy data resource integration, provide digital identification, and support archiving and preservation of datasets(Press 2004).

Metadata is also critical to ensuring that datasets will survive and continue to be accessible into the future. In distributed dataset environments, it is especially significant for making queries

due to the dispersed nature of the datasets. Effective searching is achieved by matching user queries with objects that contain identical mappings within the database. Therefore, the ability of the metadata system to efficiently improve the process of matching queries by standardizing the structure and content of indexing or cataloguing information, makes it a critical and very significant feature in the distributed database environments space (Milstead & Feldman 1999).

### 2.1.1 Components of Metadata

It is important to note that most metadata are user defined and customized to fit individual environments. However, some common components that make up generic metadata records as reported by the Federal Geographic Data Committee (FGDC 2012) and Maryland State Geographic Information Committee (MSGIC 1998) are:

- Metadata Record Information – It describes basic information about the metadata record including the language in which the record is written, a unique file identifier for the metadata record, the metadata standard used to organize the record, a point of contact for the metadata record, and the date that the metadata record written.
- Content Information – Information about the actual data set entities and attributes.
- Identification Information – Citation-level information about the data including the title, abstract, purpose for creation, status, keywords (theme and place), and extent (temporal, vertical and horizontal).
- Data Quality Information – Information about the processes and sources used to develop the data and positional and/or accuracy assessments performed.
- Maintenance Information – Information about updates to the source data, including the scope and frequency of data updates.
- Distribution Information – Information about the data distributors and methods for obtaining the data.
- Application Schema Information – Information about the schema or data models used to structure the data.

## 2.2 Data Provenance

Datasets are considered reliable if the creation process can be adequately replicated and analyzed for defects or changes to the data (Boose et al. 2007). Provenance provides information that describe the dataset creation process and all updates to the source data if any. It gives additional details on ownership of the data and data usage. As dataset increases in size and complexity, it is important to ensure that they are of good provenance. Scientific datasets are of good provenance if they contain sufficient detailed documentation to allow ease of reproduction when required (Altintas et al. 2004). An advantage of data provenance is that it helps scientists and programmers keep track of their datasets using scientific workflows despite modifications made to it during analyses, and interpretations by users. This preserves data integrity.

Combined with data provenance, metadata basically provides information about other aspects of a data such as creation of the data, Purpose of the data, Time and date of creation, creator, author or editor of data, relevant keywords, location of the data on the network, the document's usage and distribution history etc (Bruce E. Bargmeyer 2000).

There are several classes of metadata and are often considered as modules within the metadata service. They include

- **Structural Metadata:** this describes the physical and logical framework of the datasets. It is effective in managing complex digital objects and enhances adequate utilization of resources by describing how the datasets are structured and used.
- **Descriptive Metadata:** provides information pertaining to the actual content of the object, which is used to search the metadata catalog. The descriptive metadata contains direct information on an object such as title, author, subjects, keywords, publisher which allow ease of access to the user.
- **Technical Metadata**: This describes the mode of creation and storage of the datasets. It contains information such as indexes, data types, distribution lists, processes and user

security rights.

- **Administrative Metadata:** This contains information required by the repository content managers to prepare and publish datasets. It enables ease of management of the datasets within the repository. It can include information on its storage format, copyright and licensing information, and information about preservation of the datasets.
- **Process Metadata:** Process metadata is generated from the results of scientific experiments. It describe how the statistical information is processed, and can be used in determining results derived from previously executed processes.

While huge sums of scientific content are kept in distributed data repositories, the metadata service reduces complexity in storage, management and querying of these contents to facilitate reuse. Metadata standards development originated in the 1990's from the recognition of the massive potential of reusing datasets from data repositories. Since then various standards such as the Dublin core (Apps 2005), the Federal Geographic Data Committee metadata standard (FGDC) (FGDC 2012), Darwin Core Metadata standard (Tim Robertson et al. n.d.) have been introduced (McClelland 2003). Since metadata provides structured data about datasets for annotating the datasets and describe the contents, semantics and services within the datasets, it gives a foundation for easy discovery of the datasets. This has led to the increased adoption of metadata services for effective referencing and annotation of datasets stored and managed within various data repositories. Hence the need for standardization.

## 2.3 Metadata Standards

The structure of metadata varies for different scientific fields because they generally contain different sets of elements. For instance astronomical metadata will contain elements that differ from metadata associated with astrophysics datasets or molecular biology datasets and vice versa. As such it is important to standardize the metadata creation techniques. In scientific computing as in many other fields, standards are critical in ensuring uniformity and ease of

system accessibility. Conforming to standards is crucial in ensuring that datasets are accessible, easily understood and readily available.

Metadata standards provide schema's for creating metadata that allow ease of organizing information in a controlled manner by user communities. They provide common semantics for related metadata and allow for correct and proper useage, and ease of identification of the datasets by its owners and users. Metadata services and standards however complement each other because while services such as iRODS, Thredds and MCS provide interfaces for storing and querying the metadata, standards such as Dublin Core and FGDC provide a standard syntax for specifying metadata elements (Deelman et al. 2004).

Quite a large amount of metadata standards exist that provide standardized platforms for creating metadata and many more are being created everyday in diverse scientific fields. Much work is being accomplished in the national and international standards communities, especially ANSI (American National Standards Institute) and ISO (International Organization for Standardization), aimed at achieving a consensus on standardizing metadata and registries (Bruce E. Bargmeyer 2000).

A very good example is the FGDC metadata standard developed by the US Federal Geographic Data Committee which was specifically created for the implementation of a distributed discovery mechanism for national digital geospatial data (FGDC 2012). FGDC contains unique Spatial Data Organization Information and Spatial Reference Information which provide a mechanism used to represent spatial information in the data set and model of encoding coordinates in geospatial datasets respectively. By conforming to the Content Standard for Digital Geospatial Metadata (CSDGM) and also very recently the new e International Standards Organization (ISO) geographic metadata standard (19115), FGDC is without a doubt one of the most reliable and widely adopted standards for geospatial metadata.

Another standard, is the Dublin Core Metadata Standard (Apps 2005), which was developed to

for describing metadata across multiple domains. It defines frameworks for the inter-operation of metadata sets and allow these diverse types to be retrieved through a single search process (Cathro 1997). Dublin core metadata standard facilitates the development discipline-specific metadata sets that work within the frameworks of cross-domain discovery and metadata inter-operability. Dublin core provides 15 core qualifiers for creating descriptive metadata and this serves as the basic set of qualifiers for most metadata. However it is not a closed set and can be expanded to fit into different domains and datasets.

Equally worthy of note is the Darwin Core Metadata standard (Tim Robertson et al. n.d.) developed by the Taxonomic Databases Working Group. It was developed to keep metadata specifically on biodiversity information. It stores metadata on scientific data and is an extention of the Dublin Core Metadata Standard (Tim Robertson et al. n.d.). According to Cathro (Cathro 1997), the Darwin Core metadata keeps records on suggested locations for searching natural history specimen and observation databases, and recommended usage and implementations for these databases within its metadata profile. It provides suggestions for stringing queries such that they are protocol independent. It also provides guidance as to the content, structure and format of records retrieved from an information server supporting the Darwin Core. It provides a structure for mapping queries and ensuring they are are protocol independent. In general Darwin core metadata standard provides fields that guides users in identifying the content, structure and format of records retrieved from an information server (Cathro 1997).

Some other good examples of metadata standards are Ecological Metadata Language (EML), NISO Metadata for Images in XML, and Shoreline Metadata Profile of the Content Standards for Digital Geospatial Metadata (Cathro 1997). Language metadata provides specific language related meta information for content, modules, and menus of the datasets, while catalog metadata simply list the catalogs holdings for logical data items.

## 2.4 Metadata Services

Metadata services provide an infrastructure for description and discovery of datasets stored within large scale data repositories, with metadata being the main tool for achieving this. They allow scientists to record information about datasets such as information on creation, description, structure and quality of data items and to also enable query and discovery of these datasets (Singh et al. 2003). It is important to provide scalable and fault-tolerant metadata service solutions for these data repositories to facilitate dataset sharing particularly in distributed federated dataset environments. Scalability is important as it ensures that the metadata service can deal with the ever growing sizes of datasets that all need to be referenced within the metadata catalogs maintained within the metadata service. Various metadata services employ different unique structural architecture and techniques for managing its metadata and presentation to end users. These structures usually conform to various metadata standards as selected by the metadata service providers. For optimal performance, a good metadata service should provide support for as many metadata schemas as possible to enable it manage datasets generated from various sources which in most cases are compiled using varying schemas (Santos & Koblitz 2006). Usually, a common set of domain-specific metadata attributes are defined using metadata ontologies developed specifically for related application communities. For instance, geoscientists may agree on standards and formats that are useful for characterizing shared data sets and represent these using a common set of metadata attributes. On the other hand, a virtual organization that includes multiple scientific or corporate institutions may define an additional set of metadata attributes for characterizing data sets (Singh et al. 2003).

Quite a number of metadata services exist that provide a platform for managing metadata catalogs. They have been developed by different vendors over the years to satisfy the need for organizing metadata for ease of discovery and usage. Some notable metadata services used to organize metadata for scientific datasets that we will like to consider within this dissertation are:

- **THREDDS**: Thematic Real-time Environmental Distributed Data Services, Thredds is a middleware that allows transfer of scientific data from one remote location to another.

It is used to bridge the gap between data providers and data users. It provides a metadata service infrastructure that is used primarily to simplify the process of locating and accessing scientific data and also allows for direct referencing of scientific datasets using metadata. It achieves this by publishing the content of scientific data repositories with the use of a hierarchical metadata catalogue service and some other data access services, which allow users to locate and gain access to datasets based on their query. Thredds employs OPeNDAP technology as its main tool for reading datasets in several different formats and also for populating a remote database. Open Geospatial Consortium(OGC) Web Coverage Service, NetCDF subset, and bulk HTTP file transfer services are also supported in THREDDS implementation (UNIDATA n.d.). Thredds has the ability to bring together many files into virtual datasets concurrently, and reference them using related metadata tags. This significantly simplifies the process of file storage and naming, and eases the procedure of users access to large collection of files (UNIDATA n.d.) (Pandya et al. 2003). Thredds maintains its metadata within catalogs contained in xml tags and thus metadata is read using xml parsing algorithms. Thredds Data Server can be configured to work with Web Mapping Service (WMS), it can generate a WMS overlay with its data. It also provides supports for reading data via WebDAV (Zhang et al. 2009), and as such the result of queries can be displayed within a Web browser or a compatible client software.

– **iRODS**: Integrated Rule-Oriented Data System iRODS, is another open source software which is based on a client/server model. It was developed by the Data Intensive Cyber Environments (DICE) research group for metadata management for large collections of digital data distributed across multiple heterogeneous networks. Created as an improvement to DICE's storage resource broker (SRB), iRODS is a metadata service that allows users to search, access, add/extract metadata, analyze and process, replicate, and share, datasets from repositories (DICE 2009). iRODS middleware enables multiple users to locally organize and share their digital resources which are distributed physically over heterogeneous environments. At the center of the iRODS implementation is the iRODS Rule Engine which controls all operations carried out on the iRODS server which include

allowing administrators to customize and automate community-based management policies (Ward et al. 2009). It enables effective searches, management, controlling and tracking of all data access and manipulation using the iRODS iCAT Metadata Catalog, that stores state information and descriptive metadata relating to the datasets. These metadata is stored within relational database tables and iRODS provides a range of iCommands that are used to establish query and obtain access to both metadata information and the stored datasets also. While iRODS concentrates more on creating a system for management and manipulation of data sets, it pays less attention to the ordering of the metadata content and its applicability within the distributed system.

- **Metadata Catalog Service (MCS)**: The metadata catalog service is a metadata service for metadata publication, discovery and management on a file based data model. MCS uses a web service client with its metadata stored in an external catalog to accommodate metadata distributed across multiple heterogeneous metadata catalogs. Similar to iRODS, it provides a relational database for storing metadata externally. Its external catalog metadata is referenced using the external catalog name and type, and also the host name and IP address for where this external catalog may be accessed (Singh et al. 2003). The MCS is created for the data grid architecture and contains mainly logical metadata attributes that are reference datasets and enable users locate and obtain access to these datasets within the data grid. It organizes metadata as logical files stored in logical collections and provides logical views for them. A logical view contains any acyclic selection of logical files, logical collections or other logical views and access is based on authorization for every logical views (Singh et al. 2003). MCS provides support for a number of metadata standards including Dublin Core metadata standard and the Federal Geographic Data Committee FGDC, for digital geospatial metadata.
- **OPeNDAP**: The Open-source Project for a Network Data Access Protocol (OPeNDAP) technology is a framework that aims at simplifying the process of scientific data networking. The major usefulness of OPeNDAP is in allowing users to make request for data from various remote sources, making it seem like a single system. It provides a metadata service that allows referencing of the datasets using customized metadata attributes. It helps in

storing data from remote sites in whatever format is most convenient or most suitable at the site. Also in pushing data from the server to a remote location, OPeNDAP has the capacity to convert a file format to the default format on the destination repository. Apart from its usefulness in pulling data from remote databases, OPeNDAP framework can create software for pushing local data to remote locations regardless of local storage format for a wider access from prospective users (Cornillon et al. 2003). OpeNDAP also provides tools for transforming existing applications into OPeNDAP clients (i.e. enabling them to remotely access OPeNDAP served data). The clients should be able to download the data directly into the data analysis programs they are using. In ensuring global availability of data, OPeNDAP provides both client and server software, to access those data and also to make data available to remote users respectively. Some of the client software can be used to convert programs written for a data access API like NetCDF into network-ready "browsers" of remote OpeNDAP accessible data (Cathro 1997).

A variety of other metadata cataloging systems have been developed more recently with the aim of ensuring discovery and sharing of scientific datasets. The introduction of open data (Auer et al. 2007) has led to the development of various scientific metadata catalogues for dataset sharing. These systems employ various different technologies and modalities for achieving dataset archiving and sharing. Some commonly used repositories include Databib, by Purdue University and Penn State University (Michael Witt 2012), a searchable catalog/registry/directory/bibliography of research data repositories. Databib is a tool for helping researchers identify and locate online repositories of research data. Users and bibliographers can create and curate records that describe data repositories so that users can search. Another system is Open context for archaeological datasets (Kansa et al. 2007). Open Context reviews, edits, and publishes archaeological research data and archives. To preserve irreplaceable data it archives these data with university-backed repositories, including the California Digital Library. It publishes data contributed by researchers through a process that reviews, edits, and aligns data to standards. Researchers submit data for Open Context publication that document excavations, surveys, and the analyses of collections. Every record in Open Context has its own stable URL, making it easy to reference and cite very specific items of data. SIMBAD is an astronomical database for publishing astro-

nomical research data (Simbad 2013). It provides basic data, cross-identifications, bibliography and measurements for astronomical objects outside the solar system. SIMBAD can be queried by object name, coordinates and various criteria. Lists of objects and scripts can be submitted. Also we have the National Oceanographic Data Center (NODC). NODC manages the largest collection of freely available marine environmental data and information (Fiolek 2008). It holds remotely-sensed physical, chemical, and biological oceanographic data from coastal and deep ocean areas. These datasets are made available to researchers and academia on request.

A pertinent issue with using metadata services for organizing digital scientific contents is with accessibility. Most metadata service are developed as an holistic system that comes with its archiving infrastructure and also its query infrastructure. As a result, any access to the records within the metadata service is totally controlled within its system unit. For wide range collaborations, the information contained within the metadata service needs to be made available to other environments where the metadata service infrastructure may not be utilized. To achieve this sufficient infrastructure must be put in place to enable communication between the various metadata services.

There have been some efforts in the past to develop systems that integrate multiple metadata services and allow query to be made from across them. However in most of the cases found, the integrated metadata services all belong to the same architectural platform and does not encourage cross-platform integration of metadata services. For example the OpenDAP environment provides a few metadata servers such as Thredds Data Server, PyDAP, ERDDAP, OpeNDAP Hyrax, COLA GrADS Data Server (GDS) and NOAA/PMEL/EPIC Dapper Server) that can all be integrated and queried using an OpenDAP query engine (client) (Cornillon et al. 2003). Metadata is managed in XML catalogs and is read centrally from the catalogs. The OpenDap project however concentrated on format incompatibility issues and seamless data sharing to allow scientists import data directly from remote data repositories. This communication also exists solely within systems that belong within the OpenDAP server group and thus limits the datasets discovery to this environment.

The ESRI Geography network also provides a global network that allows GIS dataset providers to share their datasets with users (ESRI 2002). Using the ArcGIS system architecture, it allows dataset providers to collect, organize, manage, analyse, communicate, and distribute geographic information. For dataset users, it provides an easy way to explore, visualize, and share GIS information with visualization capabilities included (Law 2013). For communication however, every participating peer must have a GIS server before they can obtain access to the ArcGIS to share their datasets for discovery by other users using the metadata service. The participating GIS servers in turn communicate through the centralized ArcGIS for server, without which information is not possible within the ESRI Geography network. The ESRI Geography network is optimized for only GIS datasets such as spatial data, maps and geoprocessing tools.

The limiting scope of the frameworks described above provides a problem to be addressed. For larger federated curated data repositories managed over different domains, it is not uncommon to have very diverse metadata services in place. For instace a data repository can be running on iRODS, Pydap or ERRDAP metadata service while another data repository with similar datasets to share may have the Thredds, Mercury or MCS metadata service running. This presents a challenge as these different metadata services cannot communicate using their inbuilt client interfaces.

To address the critical performance issues facing metadata service integration, we introduce the VUMC a system for metadata sharing in a distributed scientific dataset environment. In the next Section, we give detailed descriptions of our solution the Virtual Unified Metadata Catalog (VUMC), an integrated metadata access interface based on the ZeroC ICE architecture.

## 2.5 Caching

Caching is widely recognized as an effective method for improving the efficiency and scalability of content delivery within search engines. It basically involves keeping fragments of frequently requested documents or links to these documents at points in the network where they can be accessed with high speed and low latency (Jelenković & Radovanović 2004). This is aimed at reducing the turnaround time for system queries by optimizing the process of locating entries and reducing the complexity involved in conducting searches. In recent times, caching has proven to be a more efficient approach to query processing especially in a distributed repository environment such as ours. Various caching mechanisms such as Redis caching adapter (Seguin 2012), Memcached (Microsystems 2008), Alternative PHP Cache(APC), WinCache (Swan 2011), and Zend Disk (Evron 2009) have been implemented in systems that manage data intensive applications.

In a metadata service caching plays an important role and actively increases the overall system performance by lowering the load on main repositories. In (Ordille & Miller 1993) for example, distributed active caching was implemented to focus queries within the repositories that contain the information being queried. By storing frequently used Metadata query characterizations closer to the user, metadata caching improves query performance significantly, eliminating constant redundant distributed communication and effectively reducing processing cost . For query processing in large scale scientific dataset repositories, cached metadata implementations help to significantly improve response time and reliability by aiding fast and efficient data discovery, and reducing the load on the network and remote database servers. System performance therefore is significantly improved using configured metadata caching mechanism by ensuring less memory usage and guarantee less strain on resources, thus ultimately optimizing query processing activities within the system. The inclusion of a metadata cache is important because it provides an easier access and location mechanism for datasets within the distributed repository environment. By archiving metadata for frequently queried datasets, the user can obtain access to these datasets more efficiently as the location within the network is already

know (Wein 2002).

In estimating the performance of a cache, two major considerations that have to be observed are the latency, and the hit rate. When talking about latency we consider the time interval for communicating with, and retrieving information from the cache. On the other hand the hit rate shows the level of accuracy experienced in retrieving cached information. The lower the latency experienced and the higher the hit rate the better the caching process. In addition some important considerations with respect to metadata caching include cache size control, collision and flushing management. Also for optimal performance, data caches should be intelligent enough to manage variations in query scenarios. This is important as query may not always be the exact same attributes as what is maintained in the cache all the time. In such cases, the cache should have the ability to perform operations such as splitting and merging of user query. In carrying out complex queries, there may be partial results in local cahe while the others are found remotely. The queries in such cases have to be treated as sub-queries which will then be merged eventually to give users a result.

### 2.5.1 Caching Algorithms

A variety of caching algorithms exist that are used to implement caching effectively and set rules required for achieving caching. They are required for determining procedures for populating a cache or cache replacements. These algorithms provide an appropriate policy for the cache placement and replacement strategies. Common caching algorithms are Least Recently Used algorithm, Most Recently Used algorithm, Random Replacement algorithm, Least-Frequently Used algorithm, Probabilistic Caching algorithm.

- **Least Recently Used (LRU) Algorithm**: The LRU is based on the temporal locality rule which states that the cached object which were not referenced in the recent past are not likely to be referenced in the near future. LRU uses the time of last reference to rate cached contents in an hierarchical structure (Vakali 2000). The most recently referenced content is placed highest on the reference table while the least recently referenced stays below the reference table. This

helps to facilitate ease of flushing at the due time. For the LRU, we make an assumption that the likelyhood for demand of a metadata record reduces with increase in time elapsed from the previous request for that record

- **Least Frequently Used (LFU) Algorithm**:The LFU algorithm performs a slightly different approach to the LRU. It is also adopts the Spatial locality rule, which suggests that if a particular memory location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future. Therefore it keeps a count of the frequency of hits on the cached contents and stores them according to this hierarchy. The record that is hit the fewest time is flushed when a new record has to be inserted.

  Other cache replacement policies include:

- **Adaptive Replacement Cache (ARC) policy:** A page cache replacement policy that does cache replacement by maintaining a recent eviction catalog of both Frequently Used and Recently Used pages plus a recent. It was developed at IBM Almaden Research Center.(Megiddo & Modha 2003)

- Most-Recently Used (MRU): MRU algorithms best use cases are in environments where older cached items are more likely to be queried for within the cache

- **Probabilistic caching**

  Probabilistic caching is an improved approach to caching aimed at reducing redundancy in cached entries. In probabilistic caching statistical approaches are employed in providing eviction probabilites that estimates when replacement should be carried out and the relevant entries to be discarded within the cache. Due to its high level of accuracy, this caching approach is increasing employed particularly in real time applications and data streaming. Various implementations of the probabilistic caching algorithm have been introduced with distinct features. The Probabilistic Shared Cache Management (PriSM) (Manikantan et al. 2012) is governed by a discrete probability distribution which computes eviction or replacement probabilities for cached entries. Every record within the cache is assigned with an eviction probability. This eviction probability is used in identifying the appropriate record for replacement in due course. ProbCache (Psaras et al. 2012) allows probabilistic distributed content caching along a path of

caches and provides an estimation of the caching capability of a cache. This is done in order to ensure adequate caching space for other ows sharing the same path, and fairly multiplex contents of different ows in caches along a shared path. Due to its effectiveness and flexibility, the probabilistic caching has also been implemented for other scenarios including in for loss recovery in reliable multicast (Feng et al. 2000), achieving coherence in volume culling for 3D streaming (Slater & Chrysanthou 1997), randomized caching in real time systems (Quinones et al. 2009) and so on.

# Chapter 3

# The Virtual Unified Medata Catalog

## 3.1 Overview

The Virtual Unified Metadata Catalog (VUMC) is a virtual implementation of a metadata service that provides a communication interface to various distributed heterogeneous metadata services to allow metadata sharing among them. The Virtual Unified metadata catalog provides a unique architecture for communicating among various existing metadata services within distributed data repositories and obtaining access to their metadata catalogs from a single unified view. The VUMC allows easy access to scientific datasets by providing tools for linking to the specified existing metadata services and querying them for existing datasets from a distinct query interface. Using the Zero C Internet Communication Engine (ICE) as its primary communication interface, it provides a remote access interface to the various data repositories and provides a framework for linking with their metadata services. Operations such as metadata query, metadata retrievals and dataset retrievals can be carried out on the metadata services integrated with the VUMC using the API's for the various metadata services. It not only provides communication to the metadata services but also provides a mechanism that maps to the query interface for the metadata services involved and allows the various heterogeneous metadata services to be queried simultaneously from a single query point as if they were a single system. Ultimately it aims to improve on the way queries are conducted to metadata services in geographically dispersed environments.

## 3.2 Basic Properties & Features

The VUMC system as shown in figure 3.1 consists of three main components that form the basic elements for metadata service communication and metadata sharing. These components serve as the main features that make up the VUMC are:

- The VUMC Server
- The VUMC Client
- ICE Communication interface

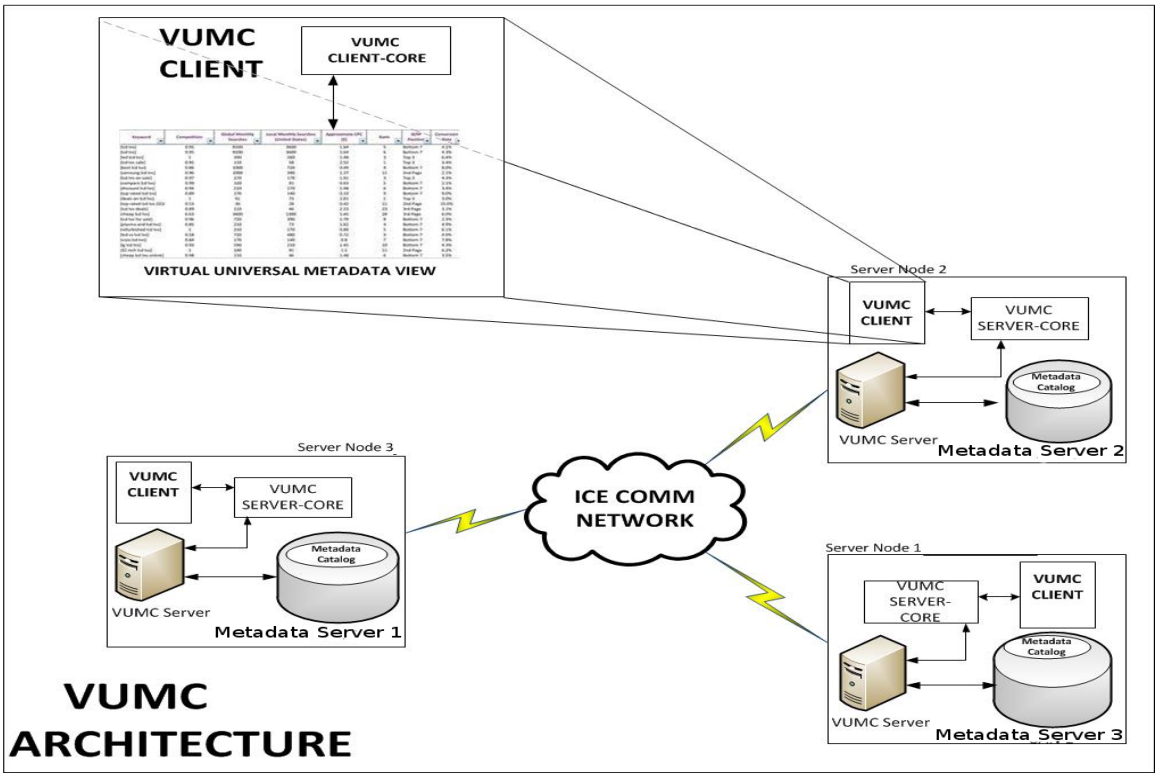


**Figure 3.1:** VUMC Architecture

The combination of these three components provide an holistic architecture for the VUMC framework. However the implementation of the VUMC server depends largely on the metadata service it will be communicating with. Details of the VUMC will be expanded upon in this section but first we give an analytical description of the structure of the system.

## 3.3 Methodology: Structure of the VUMC

With the VUMC, we introduce a unique approach for interaction between various existing heterogeneous metadata services. In achieving this, we conceptualize a distributed data repository environment which contains large scale distributed datasets located at geographically disperse sites. The sites denoted as $S_1, S_2, S_3, ..., S_n$ are autonomous and independently managed using different heterogeneous dataset management systems, and metadata management is achieved using different metadata services. As a result, each site $S_i$ contains a locally managed metadata service which could be the same as the metadata service used at another site $S_j$ but different from that at site $S_k$ as shown in figure 3.2. In such a setup, environments $S_i$ and $S_j$ may be able to share metadata due to the homogeneity of metadata service infrastructure used, however $S_k$ will be alienated from this setup as metadata sharing cannot be achieved due to the difference in its metadata service architecture.

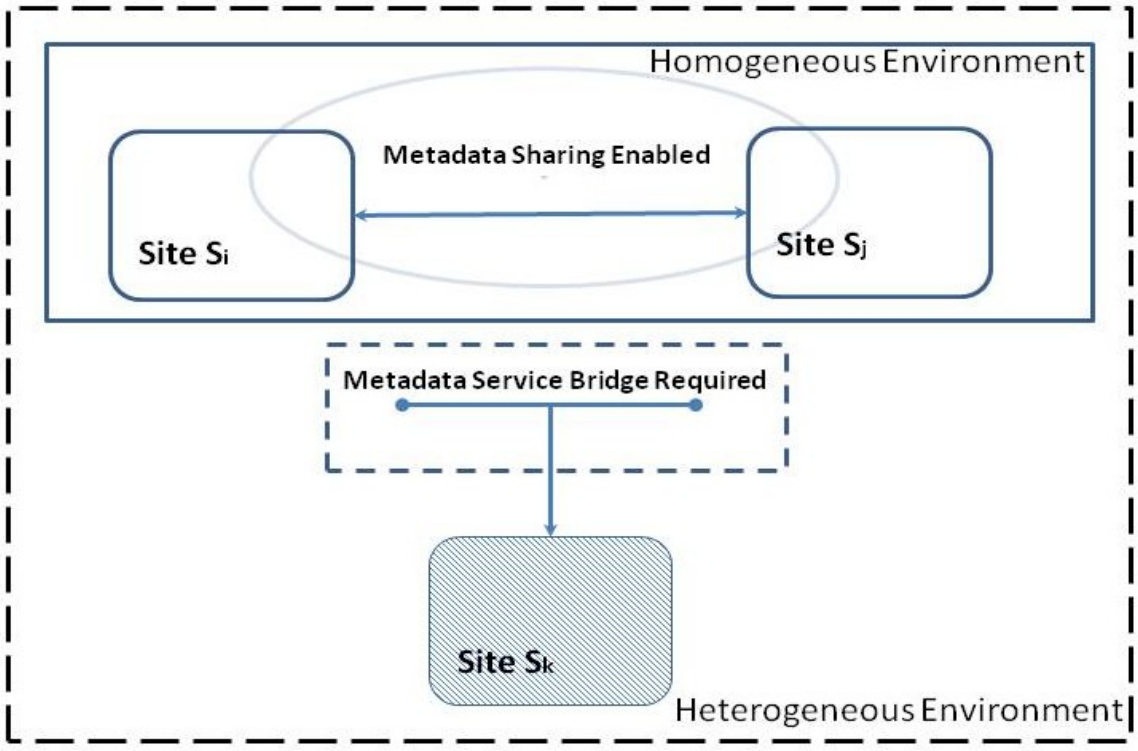


**Figure 3.2:** Incompatible Metadata Service Environment

To bridge this communication gap and allow for metadata sharing and collaboration between

all the sites irrespective of the metadata service running, we implement the VUMC system. The VUMC employs an approach of providing functional mappings for executing API function that correspond to the various metadata services which allow for remote communication, metadata query and metadata dissemination. The VUMC achieves this using a client-server architecture to provide an integrated interface for remote communication among the various heterogeneous metadata services as shown in figure 3.1.

We do not decouple or alter the metadata services in any way in the process of achieving this, but provide a unique design that uses existing API's and components of the metadata services to achieve remote metadata sharing among the different metadata services. The VUMC employs the strategic model of not altering any part of existing metadata services and this maintains its integrity as a communication medium between existing metadata services. All operations required for communication with the metadata services are carried out on the VUMC client and Server and no modification is done to the original metadata service. As a result, the system users still keep their metadata service of choice and maintain their information in their preferred format but can now effectively publish their metadata catalogs and share datasets with other existing metadata catalogs without changing their configurations. When a new metadata service is added, the details of the VUMC server attached to it are updated within the IceGrid registry and then can be easily discovered by the clients already operating within the environment. The system is optimized for metadata sharing in scientific domains which contain datasets in formats such as the Hierarchical Data Formats, (HDF/HDF5), Network Common Data Form, (NetCDF). We can now go on to describe the various components that make up the VUMC in more details.

### 3.3.1 The VUMC Server

The VUMC server is the core component of the VUMC system. It provides an interface to the connected metadata services using application programming interfaces (API's) and configuration parameters for the metadata services. Metadata services communicate with their data

repository based on a set of rules specified within the local environment. They also provide rules for metadata management and metadata retrievals for the data repository environment. These rules are organized in form of API's and the various API's provided are implemented to carry out the required operations on the data repositories. For every remote site running a metadata service, a VUMC server node needs to be implemented for achieving remote communication with other metadata services. All metadata request processing is carried out at the VUMC server side. However for the VUMC to obtain access the information stored through the metadata service, the VUMC server must be able to communicate to the metadata services. The VUMC server achieves this by linking to the communication environment of the metadata service by using its actual communication parameters within its API as shown in figure 3.3.

With the VUMC server we take advantage of the generic API's provided for the various metadata services to integrate them into the VUMC framework as shown in figure 3.1. This is achieved by including these API's for each metadata service environment within the server implementation thus allowing the server to communicate with and perform required operations to the metadata services. This allows request to be made to the metadata service and metadata query and retrievals can be accomplished. Once this is achieved, any metadata query request passed from the client to the server can in turn be communicated to the metadata service for processing using the required API's. As earlier mentioned, remote communication is achieved within the system through the Zero C ICE interface. Metadata requests is passed from the client to the server using a down-call interface that is initialized using a slice definition generated from the Zero C ICE communication interface. The server recieves the clients requests containing metadata to be queried for dataset identification and recursively queries the repositories using the relevant API's to retrieve and return the metadata information matching the query. In the same way, results of processed metadata requests are passed back to the VUMC client through an up-call interface in the opposite direction.

Each VUMC server node holds environmental parameters for the metadata service environment maintained within a configuration file. The unique parameters used include

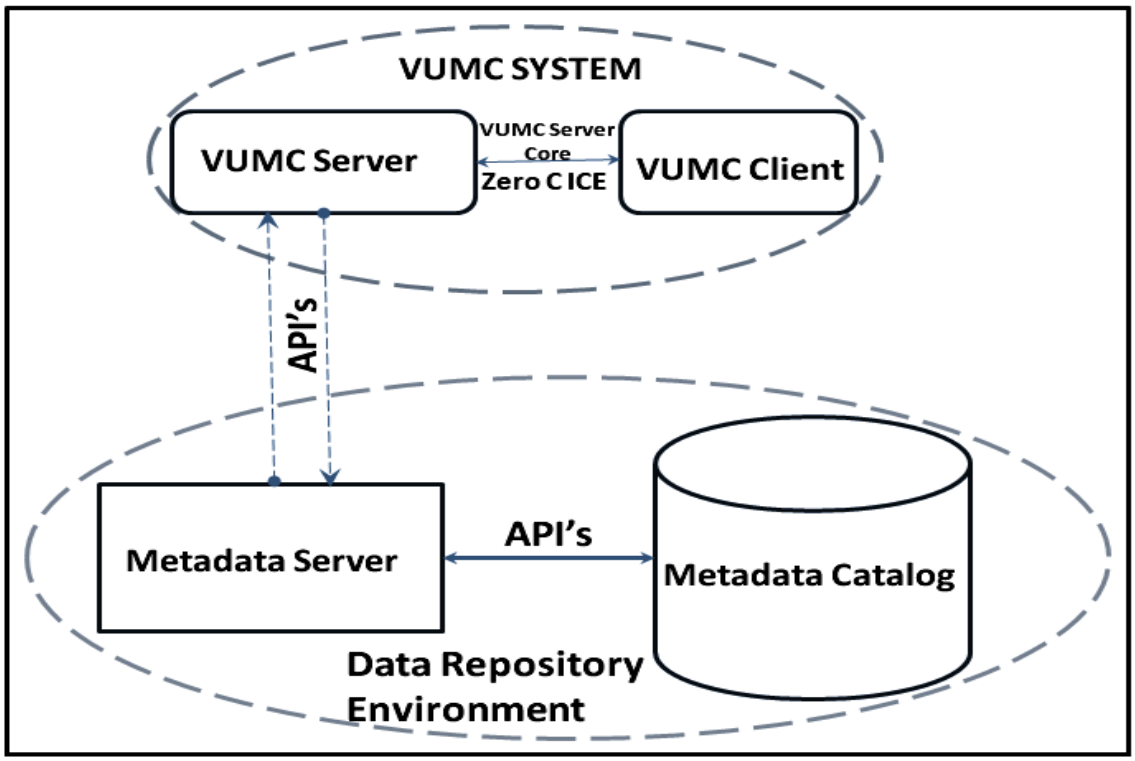


**Figure 3.3:** VUMC Data Repository Communication

- Node.Name: a unique identifier for the VUMC server node within the network
- Node.Endpoints : The communication endpoints for the server node which include the proxy address and network port number
- IceGrid/Locator: The communication address of the main server registry which allows the specific node obtain access to the general sever grid

The VUMC server node for each environment will have to be customized to suit the unique architecture of the metadata service. It is however important to state at this stage that this does not in any way change the architecture of the VUMC system or its design implementation. The unique difference is in the application code for each environment. The application code provides implementations that conform to the API's of the associated metadata service. In other words API commands for the required metadata service environment are integrated into the local enviroments server code, thus making it possible to implement operations within that environment. However for easy discovery of any VUMC server within the system, an object

proxy is assigned to the remotely located VUMC server which allows it to be easily identified and communicated with. The proxy contains the object identity of the server and addressing information for locating the server. The addressing information basically consists of the host IP address and port number on which the server is running.

**Object Adapter**: The object adapter links incoming requests to the appropriate server application code for processing of the metadata query. After the server is identified via the addressing information (i.e. Node.Name and Node.Endpoints), the query is dispatched to the appropriate implementation interface within the VUMC server through the object adapter. It dispatches the request directly to the appropriate application code which contains API's for processing the metadata request. After processing, the object adapter also coordinates the dispatch of the request output back to the originating VUMC client using the unique proxy and object adapter ID. The object adapter is made up of the object adapter ID and the object adapter name.

1. ObjectAdapter ID: The object adapter ID provides a unique identity for the object servant running at a particular server location. The object servant is responsible for carrying the user query from one point to the other within the server interface. Being able to attach a unique identity to the server is important for the query processing as requests need to be mapped to the appropriate server locations, and a unique identity distinguishes one object adapter from the other within remote VUMC environment.
2. Object Adapter Name: Just like the object adapter ID, the object adapter name also gives each environment a unique name that distinguishes it within the object adapter pool of the VUMC environment. This allows for easy identification within the VUMC environment.

For query processing as earlier mentioned, the object adapter passes the query directly to the application code from where the metadata service is queried for appropriate metadata that matches each unique query. Within the query process, three(3) main operations are performed on the metadata service being queried as shown in figure 5.2.

1. Metadata Query: The metadata query process is the first operation carried out on the metadata service when a query is passed through the VUMC server. The server upon

receiving metadata query connects to the integrated metadata services using the required configuration parameters provided and calls the predefined API's for conducting metadata queries. This process essentially searches for the location of the metadata record within the metadata service record database. If the record is within the metadata service registry, it identifies its location else it exits the process. Once the metadata record can be located within the data repository, it stores its location within an identifier for future reference. This location record is used for other operations required for dataset discovery and retrieval.

2. AVU(Metadata) Retrieval: For explicit identification of the metadata record, more information is required to assertain that a metadata record returned as result of a query is actually the required record. The AVU(Metadata) Retrieval operation essentially carries out the operation of retrieving additional information on the metadata record retrieved. For this operation, the server makes use of the metadata location retrieved during the previous query process to search through the metadata catalog for every other metadata entry linked to it. These records are retrieved and populated into a list. The list is returned to the client and displayed to provide users a detailed description of the datasets. This will improve the decision making process, as more information gathered will help users make informed decision on what record is needed.

3. Datasets Retrieval: This function calls relevant API's for the metadata services to retrieve the identified datasets. It is not always enough to identify the location of a certain metadata record. In most cases the purpose of metadata query is to identify datasets and attempt to gain access to them. The dataset retrieval operation allows users this priviledge of accessing the actual dataset for the queried metadata. Once the actual queried dataset is identified, a download link is provided accompanied with detailed metadata information. This process allows query initiators to have access to the actual files they require as a result of the query process.

In summary the operations carried out by the VUMC server are as follows.

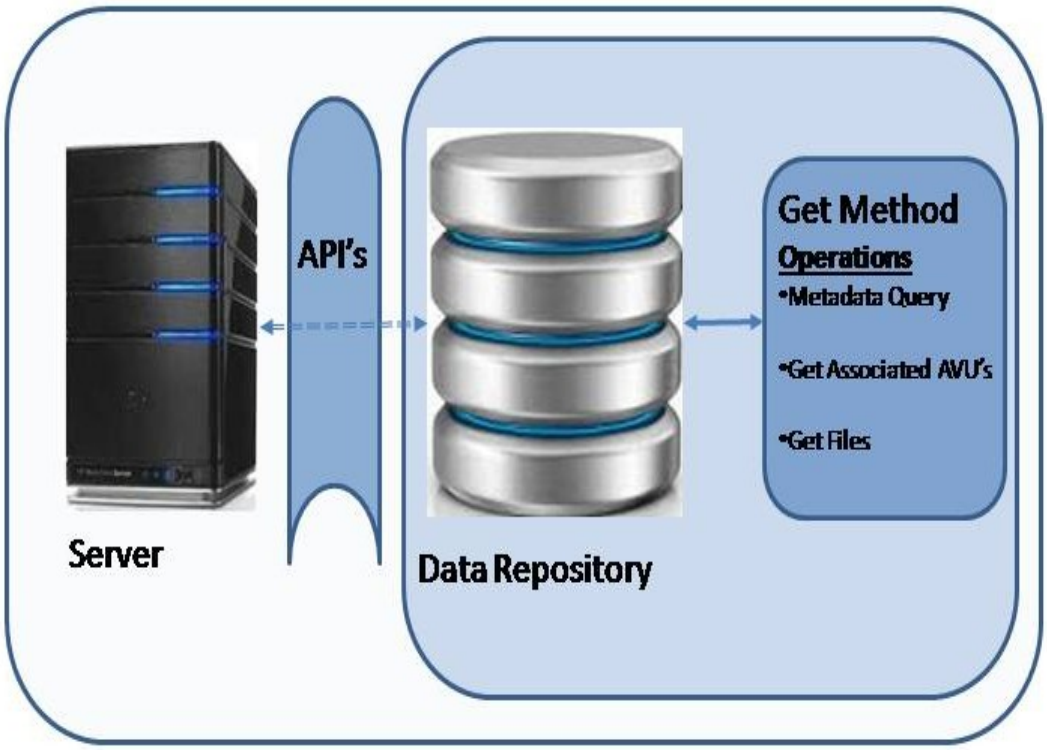


**Figure 3.4:** Server to Data Repository Communication

- Establishes communication with the metadata service using the Zero C ICE communication interface and communication API's for the metadata service.
- Recieves metadata request from client via an established down-call interface, processes the request and then sends them to the appropriate metadata service for further query.
- Sends query requests to the metadata service using the appropriate API's within the metadata service environment.
- After processing by the metadata service, the metadata generated as result of the query is sent back to the originating client via the VUMC server interface.

### 3.3.2 The VUMC Client

The VUMC client provides a communication interface for obtaining access into the VUMC system. It serves as the main point of access and entry into the system and is the sole communication interface with the system from a user point of view. It provides an initial point of query

for the system and is responsible for receiving user query for metadata. Whenever a user query is received, the VUMC client passes queries made through its interface to the VUMC server which distributes the query to appropriate sections of the system for processing. It contains fields for specifying the metadata queries and provides a down-call interface for query parsing from the user environment to the VUMC servers. The VUMC client contains an adapter interface that keeps records of server environments and allows them to be located within the environment. It uses the environmental parameters of the VUMC server residing at that remote location to keep track of the remote metadata server and communicate with it when a query request is made. For every query made, it communicates with the server registry to locate the VUMC servers using information provided within the adapter interface and connects to it using the ice grid registry, before passing the query to participating VUMC servers for processing. It then sends a communication request through to the server which opens a communication port to that server.

For communication to be made possible between the client(s) and the server(s) they must be generated from similar slice source code definitions from the Zero C ICE environment. The slice (Specification Language for ICE) definition specifies the communication interfaces, operations that can be performed on the interfaces and the data formats that can be exchanged between the client and the server. It essentially defines the modalities and conditions for client-server communication thus ensuring secure and reliable end to end communication between them. Once communication is established and the metadata request is passed, the client interface stays in an idle state until it receives a feedback from the server on the query sent. The VUMC client also provides a virtual metadata catalogue view for metadata retrieved within the distributed data repository environment controlled or linked to the VUMC system. It presents the query result using this metadata view interface which is presented via the user interface of the VUMC system. The user interface that the VUMC client employs is a java based interface that provides parameters for the metadata query and also displays the output of the user query. It is displayed on an HTML webpage hosted on apache tomcat server for easy accessibility. Additionally, the VUMC client contains a client properties file for maintaining server configuration parameters which is for referencing the required server(s) for future accessibility.

In general, the key functionalities of the VUMC client within the system can be summarized as follows

1. Provision of a communication and query interface for entry into the VUMC system
2. Accepts user query using specified AVU parameters
3. Provides an adapter interface that holds information which allows servers to be located within the environment
4. Obtains access to participating VUMC servers from the ice grid registry and passes metadata query to the identified servers for processing.
5. Hold identity mappings for the various metadata services which when sent to the server helps it identify which API's are to be called for access and metadata retrievals
6. Keeps an open interface for a feedback to the metadata query from the servers after processing
7. Upon receiving the feedback, it maps it to the user interface for display in a standardized format.

### 3.3.3 VUMC Communication Interface

The client-server communication is established using the ZeroC Internet Communication Engine (ICE). The Internet Communications Engine is a modern object-oriented toolkit that is widely used for creating realistic applications over large distributed domains. It essentially allows for creation of a communication interface among nodes within distributed and often geographically dispersed environments (Henning et al. 2009). Within the VUMC system, ICE is used primarily to achieve remote communication between the clients and servers. However, we achieve package transfer and information sharing between the VUMC clients and servers nodes using the application code we created primarily for this. Based on an object oriented architecture, ICE allows us to create object interfaces and provide corresponding implementations for these various interfaces. The interfaces corresponds to the operations to be carried out

with respect to metadata query through our system. These interfaces are defined within the slice definition. The implementations for the interfaces are on the other hand defined within our server code which will be elaborated upon in the next chapter.

### 3.3.4 IceGrid

In ensuring communication for a wide range of servers and clients that make up our distributed data repository environment, the core functionality of the ICE toolkit that we employ within the VUMC system is the IceGrid. IceGrid as described in the ICE official documentation (Henning et al. 2009) allows for communication in environments where multiple servers need to be deployed and communicated with simultaneously. It offers a wide range of features for development of applications for such environments, and also for deployment and ongoing administration of such applications. Primarily it provides some key features which serves our purpose:

1. Ensures easy server discovery
2. Activates server on demand
3. Allows for dynamic server configuration using a location service
4. Provides efficient server management with load balancing features

In addition to these features, the IceGrid provides some unique services as described in (Henning et al. 2009) which makes it a preferred option for implementation of the VUMC communication system.

- Location Service: IceGrid provides an infrastructure that allows client nodes to discover server nodes with a relative amount of ease. In cases where a required server node is not running at the time of client query, the IceGrid has the capability to activate the server automatically to allow connection as required. Also the location service within IceGrid provides features such as support for replication, load balancing and automatic fail over.
- Administrative Capacity: IceGrid provides a wide range of administrative tools that make the process of management and control of the communication interface very flexible and

reduces complexity involved. With the graphical IceGrid interface, activities such as starting a server or modifying a configuration setting are just mouse clicks away. Other tasks such as adding, removing and modifying host servers can also be easily achieved using the IceGrid administrative interface.

- Resource Allocation: System resources are efficiently managed amongst requesting clients by the IceGrid. It ensures that there is effective load balancing and clients can have equal access to resources in a structured manner based on their request conditions.
- Application Code Deployment: IceGrid makes deployment of servers less complex. It provides distinct implementation for each node to be deployed and thus avoids a lot of ambiguity and complexity involved in server deployment especially in environments where a wide range of servers are to be integrated.

#### 3.3.4.1 IceGrid Registry

The registry contained within IceGrid stores information on the various client and server nodes within the IceGrid. It maintains a persistent record of all server records and processes within the distributed environment and manages the information and server processes used in running all applications . It allows the various server nodes to be identified from within the network by keeping their environment parameters within its look up table. For explicit identification of a server node, it stores the proxy information and the object adapters credentials of the server node. The object adapter specifies the type, identity, and transport details of each of its objects and embeds the correct details when the server-side application code requests the creation of a proxy (Henning et al. 2009). Once the IceGrid is running, the registered nodes can communicate and share information amongst themselves. The nodes communicate with the registry using the IceGrid locator proxy, and at the same time the client and server nodes are lined once the IceGrid.Registry.Client.Endpoints and IceGrid.Registry.Server.Endpoints environmental parameters are set for the respective environments. With respect to the client side of the application, the main responsibility of the registry is to resolve indirect proxies as an Ice location service. By doing this it allows the client the opportunity of locating servers using

their indirect proxies. It allows clients to identify the required servers by mapping the indirect proxy links to the appropriate server node. When a client attempts an initial connection using an indirect proxy, the Ice run time within the client environment contacts the registry to convert the proxy's symbolic information into endpoints that allow the client to establish a connection (Henning et al. 2009). For access and communication, client properties "IceGrid.Registry.Client.Endpoints" which is used to locate the client and allow communication to needs a fixed port and this is pre-determined and set before connection can be achieved.

Apart from providing a look up table for registering and locating remote server nodes, the IceGrid registry also provides additional operations. It can automatically turn on server nodes when an indirect proxy is passed by a client and the resultant server is located. This effectively begins the communication process. Also with the use of indirect proxies by the IceGrid registry the client does not need advance knowledge of the address and port of a server before connection can be established. Furthermore, the IceGrid registry allows servers to migrate to different computers without the need to update proxies held by clients. The migration process is automated within the IceGrid registry lookup tables (Henning et al. 2009). An advantage of using the IceGrid functionalities is that it automatically coordinates a number of administrative operations such as replication and load balancing, error checks and publishes exceptions, Server diagnostic tests, and redundancy checks among others.

Registration of new metadata services within the IceGrid registry is dependent on the server nodes deployed within the metadata service environment. Before a new metadata service can be recognized, a VUMC server node must have been configured with the required API's for performing operations on the metadata service. The IceGrid dynamically assigns an indirect proxy for the server node which is then stored within its registry. This proxy serves as an entry point for communication to the metadata service within the VUMC environment. However operations can only be carried out on the metadata service using the API's pre-configured within the server application. We implement the IceGrid registry within our VUMC system for effective communication between our client and server nodes.

### 3.3.5 Motivation for Using ICE

Zero C Ice provides a platform with a high threshold of efficiency and effectiveness in provisioning object-oriented middle-ware applications suitable for use in heterogeneous environments. it contains a full set of features that support development of realistic distributed applications for a wide variety of domains with mappings for various environments. The heterogeneity functionality availed with the use of this platform makes it stand out for us as a preferred choice. Also the Ice platform provides less complexity and is easy to adapt and use. It is and created to achieve avoidance of unnecessary complexity in application design. Some other distinct qualities that make us opt for Ice include:

- Security: The importance of ensuring security of metadata access cannot be over emphasized. Unauthorized access to published information will be a major bottleneck in this setup as sensitive metadata information and datasets might be accessed unlawfully. Ice provides an implementation that has built-in security, making it suitable for use over insecure public networks. Ice uses a firewall traversal service to allow clients and servers to securely communicate without compromising security. Client- server traffic is SSL-encrypted using public key certificates and is bidirectional. It offers support for mutual authentication as well as secure session management (Henning et al. 2009) . The ability of Ice to provide us this functionality as a part of its integrated architecture makes it a preferred choice in developing our system as against other distributed communication platforms.

- Network and System Usage: In enabling communication in distributed environments, Ice provides functionalities that manages network bandwidth, memory use, thus reducing CPU and system overhead. In building reliable applications, it provide an implementation that is efficient in network bandwidth, memory use, and CPU overhead (Henning et al. 2009).

- Reliability: It has also recorded major successes in building several communication channel frameworks and has thus been adopted by numerous organization that require complex distributed communication models. Some notable companies that have adopted this technology

include NASA, Boeing, L-3 Nautronix Australia, CCTVnet, DATADVANCE Russia, just to mention a few (ZeroC 2013). This track record goes a long way in validating its effectiveness and gives us a basis for exploring this middle-ware software platform for developing our application.

- Version control: When using commercial products such as Ice it is inevitable that version upgrades will be introduced from time to time. The pertinent question this raises is on how this may affect the performance of the VUMC system. Ice was developed with this put into consideration. Version upgrades are organized in such a way that they do not change an existing type system but only extend it instead (Henning et al. 2009). This allows for backward compatibility between new versions and existing versions of Ice because newer versions are just an extension of features of the older version and not a complete replacement. As a result all Ice functions used in development of our system are not directly affected whenever a system upgrade of Ice is done or when a new version is released. However like in any other application, improvements have to be made to the VUMC from time to time and these updates will take into consideration improvements in the version of Ice.It is important to know that to integrate updates into the existing system none of the existing definitions have to be touched. This is because Ice provides a mechanism for implementing multiple interfaces for a single object. As a result a backup interface can be updated before it is updated on the main interfaces. Updates can also be rolled back in case of errors to avoid system downtimes during the update process. All VUMC client-server environments can subsequently be notified of version updates using a publish and subscribe feature within the Ice toolkit.

## 3.4 VUMC Caching Infrastructure

The VUMC provides a unique system for metadata service catalog integration, and information sharing among them. However, we identified a problem area with respect to query management and dataset retrievals which we decided to address. We will like to consider an improvement area in the query processing architecture of the VUMC system. As earlier stated, the query processing system controls all activities involved in achieving dataset retrievals within the VUMC.

The normal query process is iterative and is executed by passing the metadata query request to all metadata services within the system for every query initiated. For every metadata query request passed through the client via the user interface, the VUMC conducts remote query to all available servers within the network for query of its metadata service. For instance, for a query $q_1$ made from site $S_i \in [S_1, \ldots, S_n]$, the query processing system sends the query attributes to the VUMC server running on the sites $[S_1, \ldots, S_n]$. The servers then probe their metadata services using the API before returning the feedback to the site $S_1$ where the query was initiated from. After the query process is completed, a set of results $R(r_1^1, r_2^1, r_3^1 ..., r_n^1)$ matching the query $q_1$ that correspond to each site within $[S_1, \ldots, S_n]$ that returns a result for the query is returned.

This process requires a long time to query all available sites running metadata services especially within very large distributed dataset environment. If the same query was made few hours later, the initial process will be repeated all over again before a feedback is sent to the client again. In a scenario where the same metadata query is often made from a particular site, it becomes expensive to carry out this process continuously every time the query is made. The cost of carrying out all these operations will over time have an adverse effect on the system and generate a lot of redundant queries which causes a lag on the system. We also see this as being very inefficient, time consuming, results in bandwidth wastage, high latency and allows for a lot of redundancy in processed queries. The issue of turn around time can also not be over-emphasized. With a more predictive architecture, queries can be narrowed to areas or concentration to avoid ambiguous searches and minimize wastages in system resources and cost of system queries. It is therefore important and very practical to have a system in place that can intelligently carry out queries to avoid the problems stated above. Such systems will be very beneficial, particularly to cater for frequently occurring queries. We therefore decided to look at techniques for providing a more efficient solution for performing queries within the system.

In optimizing query processing, various techniques have been employed in the past which have been found to be not so efficient and overtime have been replaced with more efficient techniques.

In the past, a commonly adopted approach was using architectures that maintain datasets in a single centralized location from which it can be easily queried. As such, new datasets have to be stored on the central server to make it available for access. However this method is inefficient and is not very scalable, with high system overheard and high network latency. Thus distributed architectures are more widely adopted due to heterogeniety and distributed nature of most dataset sources (Schwarte et al. 2011)(Gehani et al. 2010). Replication has also been considered as an option but largely results in increased storage cost, and redundancy as most replicas are rarely accessed. It is therefore concieved that an intelligently designed persistent cache will be a more economical solution for query processing in data grids (Ahmed et al. 2005).

Caching provides a more reliable approach to query processing within the system. It becomes necessary to perform remote caching to allow remote queries more efficiently done in a practical way. It will significantly reduce the load on the system while it also improves response time and reliability of query processing (Leonard & Albee 2010). Furthermore, it improves performance as it reduces remote query thus avoiding increased latency within the system. With its implementation the VUMC will not have to perform a universal search for every query requested but can simply look up the cache and in case of a hit streamline the query to specific locations within the network.

With this in mind, we implemented the design of an integrated caching algorithm for the VUMC system. This helps in improving and optimizing the performance of the VUMC query processing system by achieving the following

- Integration of an embedded caching mechanism for metadata information on frequently queried datasets that allows ease of datasets location using the cached information.
- Probabilistic caching algorithm to predict the probability of reoccurence of a query before caching the result.
- Integrate of a memory location that serves as a first point of query within the system and reduce latency using intelligent algorithms for caching and query processing.

- Study of system performance of our caching mechanism in a simulated experiment and show that our approach guarantees improved performance in query processing.
- Measurement of improvements in query time, latency experienced in query processing, and accuracy of the system after the implementation of the integrated caching mechanism .

## 3.5 VUMC Cache Methodology

Within the VUMC environment, we envisage that various queries for metadata may be made repeatedly from the same remote site where a current query originates from. For instance if a query, $q_1$ is made from site $S_1$, there is a possibility that such query will be made again from that site within a short time space. Prior to the cache implementation, for every such query, the system had to search through all remote metadata servers to attempt to locate a matching record for the query. This method is very inefficient and results largely in latency issues and network overhead as earlier mentioned. It became important to have a form of predictive architecture to envisage the likelihood of repeated queries from a site and make provision for it. Hence the introduction of the VUMC cache.

Our solution provides a specialized caching system for the VUMC to allow local caching of frequently queried metadata, thereby allowing for a more optimal approach to query processing within the VUMC system. The VUMC cache is implemented on the VUMC client environment. The VUMC cache stores state information about a query result which allows faster query processing within the VUMC system. For relevant metadata result received by the client, a write request is sent to the cache and relevant information for identifying the dataset is stored within the cache. The information stored include the AVU parameters, object adapter address and the adapter identity for the remote server. However we make use of a probabilistic driven caching algorithm in determining what records are to be kept within the cache for future referencing. For every query initiated from a VUMC environment, the local cache is first queried for matching records. If there is a hit, the cached information is retrieved and the object adapter ID and

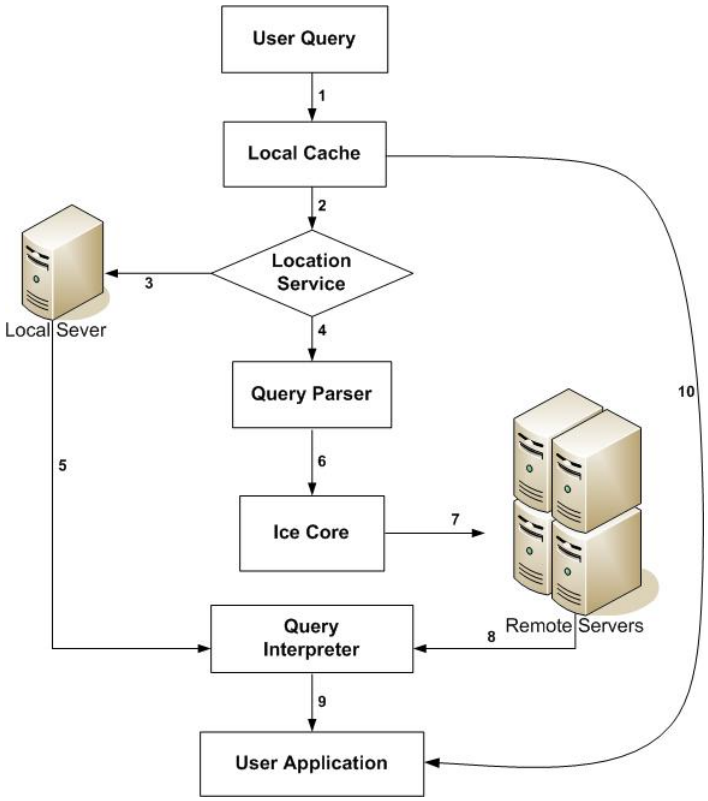


**Figure 3.5:** Cache Flow

address is used to locate the metadata server within the network. Once the server is located, the query is sent directly to it for processing and the result can be obtained. This provides a faster query process as against if all participating servers were to be queried one after the other.

#### 3.5.0.1 Caching Process Flow

The query process with the introduction of the cache is now extended with additional steps which are detailed below as described in Figure 3.5.

- **Step 1:** For a query made by the user, the query is passed to the local cache to probe for matching records. This is the initial step which will determine the direction the metadata query will proceed.
- **Step 2:** The local cache is examined for records that correspond to the query parameters. If a matching record is found, there is a hit and the cache provides the proxy and address

information for the server where the record is located. It the record is not contained in the cache, the query returns null. The result is then passed to the location service.

- **Steps 3 & 4:** The location service receives the result of the previous operation and performs one of the following tasks.

  1. If the process returns null, it send the oringinal query parameters to the query parser to probe all servers environments for matching records for the query.
  2. If the cache returns a value, the location service checks for the destination address of the server. For a local address, it passes the query to the local server for processing, and in the same way for a remote address, it passes the query to the query parser for identification of the server location and distribution of the query as appropriate.

- **Steps 5:** For queries forwarded to the local VUMC server, the server processes it by querying the metadata service in its environment. After the local query process, the retrieved metadata records are parsed to the query interpreter for final processing before it will be transferred to the user application.

- **Steps 6 & 7:** The query parser is responsible for filtering and mapping the AVU query attributes to the respective remotely connected server environments within the system. The ICE grid registry within the ICE Core identifies the corresponding remote server and allows communication with the server as appropriate.

- **Step 8:** The remote VUMC server at this stage performs the query on its metadata service and passes the result to the query intepreter.

- **Step 9:** The Query interpreter collects the results of processed metadata requests from the corresponding servers (local and remote servers), processes and sends its to the user interface at the application layer for presentation to the user.

- **Step 10:** The final output is displayed via the user interface of the user application. The view provides a detailed list of metadata records to the users as well as a link to the actual referenced file for download if required. The metadata retrieved is also sent back to the cache to update the cache records for future query reference. This process keeps the cache

up-to-date and prevents high miss rate in cache query.

# Chapter 4

# Analytical Model for the VUMC

In validating the VUMC model presented, we did a careful estimation of the key elements of the VUMC system and its architecture, with respect to common related mathematical models. It is important to consider common factors that may affect the system when estimating its performance and also provide schemes to validate the architectural set up of the system. We present an analysis of the VUMC, evaluating the time of query and the hit rate accomplished when querying metadata services. We also consider the effects of implementing the VUMC cache by analysing the methods for probabilistic caching, analysing the expected delay experienced in querying the cache, analysing the hit rate after the implementation of the cache and expressing how the optimal cache size is determined.

## 4.1 VUMC System Analysis

### 4.1.1 Analysis of Query Process

We consider a distributed computing environment that contains a collection of $n$ remote sites, with each site running different metadata services. Each metadata service contains various metadata records which could either be replicated over several sites or unique to a particular site. All remote sites in this set up contain both the VUMC client and VUMC server. The servers receive frequent communication from the clients with various concurrent queries for resources held at the various servers. We provide mathematical notations to express the relationship and describe how the query operations are carried out within this environments.

We first show the expression for a single client query made and then extend it for multiple concurrent queries made.

For a remote query $q_1$ made from a remote site say $S \in S1, ..., Sn$, all participating server environments are queried and the client receives the response from all servers containing matching records. That is response $r_1$ from server $S_1$, $r_2$ from server $S_2$, and so on, until it receives from the nth server which is the last server containing matching records within the distributed environment. This result is sent back to the client where the query originated from. Therefore the query result received from the participating sites for query $q_1$ will be

$$\text{Query result } q_1 = r_1^1, r_2^1, r_3^1, ..., r_m^1$$

This expression shows the result received from a single clients query to the resultant VUMC environment; where $r_1^1$ is the result of the query $q_1$ to a unique site $S_1$; while $r_2^1$ is the result of the query $q_1$ to a unique site $S_2$ and so on up till the result from site $s_m$; $r_m^1$ is retrieved. The result generated is expressed as the Query result $q_1$. The results can then be analyzed by the user before selecting the closest matching record on the metadata information retrieved.

A set of multiple concurrent queries on the remote sites from various clients will generate a finite set of results corresponding to each individual query made. For instance consider a scenario where a query $q_1$ is made by a client $X$, and query $q_2$ is made by a client $Y$, query $q_3$ by another client $Z$, ..., till we have query $q_n$ made by the last client. For $m$ remote sites, a set of $n$ queries will generate the following results after processing.

$$\text{Queries Q}(q_1,\ q_2,\ q_3,\ ...,\ q_n) = [r_1^1, r_2^1, r_3^1, ..., r_m^1] + [r_1^2, r_2^2, r_3^2, ..., r_m^2] + [r_1^3, r_2^3, r_3^3, ..., r_m^3], + ... \\ +, [r_1^n, r_2^n, r_3^n, ..., r_m^n]$$

In the above expression, the parameters ($q_1$, $q_2$, $q_3$, ..., $q_n$) represent the various concurrent queries made by existing clients in the environment. The result for each query is presented as $[r_1^1, r_2^1, r_3^1, ..., r_m^1]$ for query $q_1$, $[r_1^2, r_2^2, r_3^2, ..., r_m^2]$ for query $q_2$ and so on until all the queries made are accounted for. The expression above shows the total load expected in response to concurrent queries made at a particular time, T within the environment. This is expressed as a summation of all the query responses received for the concurrent queries made. This can be

used to estimate the load experienced within the system during query process and can help to keep track of system performance.

### 4.1.2 Analysis of Query Time

We analyze the time cost involved in querying the remote sites and receiving feedback for the search. This evaluation considers the time difference between when the queries are sent from the participating client environments, to the time the query result is returned to the various requesting clients.

The time difference annotated by $t_i^j$ is the time associated with a query result $r_i^j$, from a site within the distributed computing environment, where i is the query identifier, and j signifies the site from which the result, r is retrieved. However, since results for each query is recieved from multiple sites, the time for retrieving the results from the remote sites needs to be carefully considered in estimating the overall query time. As a result we estimate the average time for conducting searches using the time for each result generated for the respective queries. Since the time for retrieving the records from each site can be easily estimated using $t_i^j$, the average time for the query from all remote sites can be evaluated by getting a summation of the query time from each site, and evaluating against the number of queries results retrieved. Average Query Time of $q1$, expressed as $\theta(t)$ is

$$\theta(t) = \frac{\sum_{j=1}^{m} t_i^j}{m} \tag{4.1}$$

where

$\theta(t)$ is the notation for average time for a single query, if $t$ is the time

$t_i^j$ which is the time associated with obtaining a query result $r_i^j$ is computed by $t_j^i = t_1^1, t_2^1, t_3^1, ..., t_m^1$,

$t_1^1$ represents the time taken for the response to the query $q_1$ to be retrieved from site $S_1$; $t_2^1$ is the time taken to obtain a response from site $S_1$; $t_3^1$ is the time taken to obtain a response from site $S_3$; ,..., and $t_m^1$ is the time taken to obtain a response from the last queried site $S_m$

$m$ is the number of query results retrieved,

$i$ corresponds to the query identifier (i.e query $q_i$), and

$j$ is the site from which the query is retrieved.

The average query time $\theta(t)$ therefore provides us with a deterministic estimation of the time for retrieving a specific record from the distributed computing environment.

For multiple concurrent queries, the time cost can then be computed as the summation of the average time cost of all individual queries made simultaneously. This can be expressed as

$$Totalcost, \theta(T) = \theta(t_1) + \theta(t_2) + \theta(t_3) + \ldots + \theta(t_n)$$

where $\theta(T)$ is the total cost experienced during simultaneous system queries from various client environments

$\theta(t_1)$ is the cost estimate for a query from client $C_1$, $\theta(t_2)$ is the cost estimate for a query from client $C_2$, $\theta(t_3)$ is the cost estimate for a query from client $C_3$ and $\theta(t_n)$ is the cost from the last client $C_n$.

The total cost estimation gives us an idea of how long it a group of concurrent queries will be running within the system. However other considerations need to be put in place. Factors like network latency, propagation delay which is experienced as a result of information being transmitted over a network and system downtimes may affect the result of this computation. Therefore they also need to be considered for the networking environment.

### 4.1.3 Analysis of Hit Rate

The hit rate checks the probability that a record would be discovered and retrieved every time a query is made with the same information. This checks the reliability of the query engine to perform a thorough query and retrieve every matching record possible within the distributed computing environment. Some considerations however with respect to the hit rate include the quality of the query attributes as supplied by the user and the availability of a reliable communication channel to all possible remote sites at the time of the query. Another consideration is with how the remote sites are organized and managed. In the case of an update to the remote metadata repository, a record might have been deleted or edited with new varying

parameters. This can obviously change the results of the query even if the same search criteria are used. With all above conditions met, the hit rate can be defined as follows.

In the event of a query $q_1$ occurring $\beta$ times, if a result $r$ occurs $\alpha$ times, where $\alpha \leq \beta$, the hit rate is the probability that $\alpha$ is constant for every occurrence of $r$ provided $q_1$ is constant (Thaler & Ravishankar 1998). Therefore for various queries of a particular record q1, the average hit rate of q1 is expressed as

$$HitRate, H = \frac{\sum_{j=1}^{m} \alpha_j}{r\beta} \tag{4.2}$$

(Mookerjee & Tan 2002)

where

$H$ is the Hit Rate

$m$ is the number of query results

$\beta$ is the number of times the query is carried out

$r$ is the query result received

$\alpha_j$ is the frequency of occurence of the queried parameter $j$

### 4.1.4 Expected Number of Simultaneous Queries

For every distributed repository environment, an important attribute that plays a big role and needs to be considered is the maximum volume of queries that will be carried out on the system simultaneous at every point in time. This serves as a basis for making critical decisions with respect to the maximum load expected, required resources for effective management of these requests, and system requirements to suit unique environments.

In estimating the number of simultaneous queries expected, we must consider two main parameters which are the total number of times queries are carried out, and this is evaluated in relation to the frequency of occurence of that record or similar records within the output generated. This is computed as

$$ExpectedSimultaneousQueries, H_0 = \frac{1-\alpha}{\beta}n \tag{4.3}$$

(Mookerjee & Tan 2002)

where

$H_0$ is the number of simultaneous queries expected n is the number of available records to be queried

$\alpha$ is the frequence of occurence of the records within the output

$\beta$ signifies the number of times the query is performed

The ratio of the effect of the frequency of generated outputs and the number of times related queries are performed with respect to the total number of available records gives us an estimation of the volume of queries that hit the system periodically. This estimation as previously identified is a key attribute to understanding the performance of the system.

### 4.1.5 VUMC Cache Analysis

In the design of the cache, and provision of suitable caching policy to match the system, some factors must be considered which play a key role in the system performance.

- the cache size,
- sizes of metadata to be cached
- Ages of cached metadata

We provide the following expressions to define how the cache is organized for the VUMC and the caching policies employed.

For every query $q_i$ made, a set of results $(r_1^i, r_2^i, \ldots, r_m^i)$, is generated from metadata services on sites $S_1$ to $S_m$ respectively. We assume that the sizes of the results $r_1^i, r_2^i, \ldots, r_m^i$ generated from this query are $\lambda_1^1, \lambda_1^2, ..., \lambda_m^1$. Suppose a local cache has a size of $L$, and is filled with $n$ metadata files. If the metadata size of a set of queries $Q(q_1, q_2, ..., q_n)$, made at a site are $\lambda([\lambda_1^1, \lambda_2^1, ..., \lambda_m^1], [\lambda_1^2, \lambda_2^2, ..., \lambda_m^2]), ..., [\lambda_1^n, \lambda_2^n, ..., \lambda_m^n]$; where $[\lambda_1^1, \lambda_2^1, ..., \lambda_m^1]$ is an expression of the

size of results received for query $q_1$ from all remote sites, $[\lambda_1^2, \lambda_2^2, ..., \lambda_m^2]$ is the size of results received for query $q_2$ from all remote sites and $[\lambda_1^n, \lambda_2^n, ..., \lambda_m^n]$ is the size of results received for query $q_n$ from all remote sites then,

$$\sum_{i=1}^{n}\sum_{j=1}^{m}\lambda \leq L \tag{4.4}$$

(Mookerjee & Tan 2002)

Where

$L$ is the total cache size, and

$\lambda$ represents the size of all results generated as explained above.

This shows that the sizes of the results generated from all query must be less than the total cache size for it to be ingested within the cache. It must be stated at this point that a younger cached metadata record has a higher priority than an older record within the cache. Their ages correspond to the amount of time spent within the cache.

#### 4.1.5.1 Probabilistic Caching Expression

For a record stored within the cache, we provide a probabilistic notation that suggest conditions for caching the records. The condition for caching the records is dependent on the possibility that a subsequent query will be made for that record within a short space of time. Caching such records reduces the turn around time for remote metadata queries, reduces network overhead and allows for faster query processing. A probabilistic approach achieves better performance as it allows caching of records that are often required as against randomly caching of any records that result from the query.

We declare the average ages of a cached record (i.e. the average time a cached record spends within the cache) corresponding to the queries $Q(q_1, q_2, ..., q_n)$, as $G(g_1, g_2, ..., g_n)$, then for each query $q_i$. Let the number of ordering of ages in query $q_i$ be $n_i$. Given that query $q_i$ has ages $(g_1^i, g_2^i, ..., g_m^i)$ the probability density that query $q_i$ will be repeated is expressed as

$$P(q_i) = \prod_{j=1}^{m} \frac{\theta(g_j^i)}{H_{\theta_j}} = \prod_{j=1}^{m} \frac{\beta}{1-\alpha} \frac{\theta(g_j^i)}{n} \tag{4.5}$$

(Mookerjee & Tan 2002)

where

$m$ is the number of query results.

$\beta$ is the number of times the query is carried out.

$\alpha$ is the frequency of occurrence of a record within the output.

$\theta(g_j^i)$ is the average ages of a record within the cache.

$H_{\theta_j}$ is the expected number of cache hits per unit time.

This equation expresses the probability that the query $q_i$ will be made again from the same location within a specified time frame. The probability ratio is used to determine if the results recieved for the query can be cached based on this probability rating. Suppose the cache policy has a threshold probability of $\delta$, then the result $(r_1^i, r_2^i, \ldots, r_m^i)$ corresponding to $q_i$ will only be cached if the condition $P(q_i) \geq \delta$ is met, that is the probability ratio is higher than the threshold value. This gives a basis for expecting a subsequent query for that record and storing it within the cache.

#### 4.1.5.2 Analysis of Expected Delay

With frequent access to the cache, there exists a measure of delay involved in retrieving records kept within the cache. The delay can be attributed to various factors. For every hit to the cache there is a time lapse experienced which ultimately affects the performance of the system. It reflects in the latency of each query process done through the cache. Also propagation delays due to network resources and environmental factors within the network must be considered. If the ages $g_1, g_2, ..., g_m$ of query results $r_1, r_2, ..., r_m$ satisfies the condition that $g_1 \leq g_2 \leq ... \leq g_m$,

then the expected delay, D per access for $m$ query results retrieved is given by

$$D = \frac{1}{m}\sum_{i=1}^{m} d_i - \sum_{r=0}^{m} d_r p_r y_r \tag{4.6}$$

(Mookerjee & Tan 2002)

The first iterator in equation 4.6 is the average delay experienced in locating a metadata record before the implementation of the cache. On the other hand the second iterator calculates the savings accrued after implementing the cache. This gives us an estimation of the overall delay experienced in the query process after caching, where

$d_i$ is the average delay for the $ith$ metadata record retrieved

$d_r$ is the average delay experienced per metadata record retrieved.

$p_r$ is the probability that there are $r$ records within the cache, and

$y_r$ is the probability that one of the $r$ records in the cache will be accessed.

The delay experienced is computed by performing a comparism of the average delay experienced in retrieving the $ith$ metadata record in relation to probability that the record exists within the cache and the probability that any cahed record will be retrieved. The number of query results generated is also required in estimating the average delay experienced as shown. Advanced details on this can be found in (Mookerjee & Tan 2002)

#### 4.1.5.3 Analysis of Cache Hit Rate

Estimation of the hit rate helps to keep track of the performance of the cache with respect to delivery of query results. The higher the hit rate, the better the performance of the cache. The cache hit ratio measures the probability of access of one or more of the records that are contained within the cache. An ideal scenario is one in which we get a cache hit of 1 for every query made. However in most cases a hit rate above 0.5 is considered a good result. The hit ratio per single query is given by

$$h = \sum_{r=0}^{m} p_r y_r \tag{4.7}$$

(Mookerjee & Tan 2002)

However for a set of multiple concurrent queries carried out on the cache, the aggregated hit ratio is given by

$$H = \sum_{y=0}^{n} \sum_{r=0}^{m} p_r y_r \tag{4.8}$$

(Mookerjee & Tan 2002)

where

$P_r$ is the probability that there are r records within the cache;

$Y_r$ is the probability that one of the r records within the cache will be accessed

This performance estimation shows us how effective the cache is for the general query process.

# Chapter 5

# The VUMC Implementation

This chapter details the core architectural implementation and the design of the VUMC system. We will also discuss in more details the process of integration and communication with metadata services, and how metadata queries are carried out on these metadata services.

## 5.1 VUMC Implementation Infrastructure

The VUMC provides two different implementation environments for the VUMC client and the VUMC server respectively. They are separated by implementation but linked via generic interfaces declared within the Slice API definition. The Slice API definition is a purely declarative construct which is not language specific. It provides a mechanism for declaring the various interfaces used for communication between the client and server. The description or declaration of these interfaces within the Slice API definition does not include its implementation which is contained within the client and server code respectively. Asides interface declaration, Slice API also largely deals with declaring data types that will be used in the application to be developed. Data type declaration is very important in creating our Slice API definition because data exchange between client and server can only achieved if their types are pre-defined in Slice API. This helps to ensure the language independency of the developed application. The well-defined Slice API definition can however be compiled to run in any language environment it is required for. Ice defines language mappings for C++, Java, C#, Python, Objective-C, Ruby, and PHP.

### 5.1.1 VUMC Slice API Definition

For our Slice API definition as described in figure 5.1, we start by declaring the basic metadata query attributes which essentially are the data types and variables required for our VUMC application. The first we declare are the attribute, value and unit of type string which are the metadata query attributes. They provide generic identifiers that correspond to the standard for maintaning metadata in most existing metadata services which are ordered as key-value pairs. The attribute variable corresponds to the attribute name of a metadata record, the value is the corresponding value of the attribute, while the unit is the unit associated with the value wherever necessary. Some records require a unit of measurement while others do not. As a result the unit is often an optional feature used during query but is also equally important.

```
module Avu
{
    struct Triplet
    {
        string attribute;
        string value;
        string unit;
    };

    dictionary<string, string> FileList;
    sequence<Triplet> TripletList;
    dictionary<string, TripletList> FileAvus;

    interface Triplets
    {
        void getIrods(Triplet params, out FileList files, out FileAvus avus, out string status);
        void getThredds(Triplet params, out FileList files, out FileAvus avus, out string status);
        void getMcs(Triplet params, out FileList files, out FileAvus avus, out string status);
    };
```

**Figure 5.1:** VUMC Slice API Definition

Next within our Slice API definition, we declare variable that will hold query results after processing and their types. These also need to be declared within the Slice API definition

as they will be used system wide. The first is the *FileList* which holds the metadata record location retrieved after the metadata query operation. Next to that is the *TripletList* which will store the retrieved metadata list from the getAVU operation carried out by the VUMC server. Next we declare the *FileAvus* which holds the actual files location that correspond to the metadata being retrieved. From here the files can be accessed and retrieved where necessary.

Finally we declare the interfaces we will be implementing within the VUMC system. At the moment, the VUMC implements three interfaces for the three metadata services it can communicate with (i.e. *getiRODS*, *getThredds* and *getMCS* for the *iRODS*, *Thredds* and *MCS* metadata services respectively). The interfaces are methods that take four variables each which correspond to the operations that can be implemented on the interfaces. It is important that the interfaces are well defined at this stage as this impacts on all operations that can be carried out via these interface throughout the system.

## 5.2 VUMC Client & Server Implementation

### 5.2.1 VUMC Client Implementation

The VUMC client was created using java mappings for the ICE environment. It performs two main broad functions.

1. A communication interface for metadata query passing to the VUMC server
2. A direct link to the user interface for accepting metadata query attributes and display of processed metadata.

From the Slice API definition compilation, the client contains interfaces generated for remote communication with configured VUMC servers within the VUMC system. For communication with the VUMC system, the client links up using the ICE interface. The VUMC is configured for bidirectional connections between the server and client. It links to the server environment through the Icegrid registry by maintaining the registry information in a config.client config-

uration file. A "client.properties" configuration file is also provided that optionally contains details of specific server addresses that a direct communication link is required to. Once communication is achieved, metadata query can then be passed through from the user interface query to the server environment. The client at this point goes into an idle state while it waits for the query response from the server environment.

The second part of the VUMC client is the user interface. The user interface was created using JavaServer Pages (.jsp) Technology. Metadata query is accepted via this interface and the result of the query is also published here. Using html tags, required fields are provided for accepting AVU (Attribute, Value, Unit) query as specified within the Slice API definition. It can accept single AVU query parameter or a combination of AVU query parameters for querying various categories of metadata records managed with any of the different integrated metadata services. Once the query is submitted, it is passed through the ICE communication environment to the server side for processing. The output of processing after it is returned needs also to be well organized at the presentation layer for easy readability. For display of metadata query output, dynamic tables are generated to display the output in a structured format. The display conforms to the dublin core metadata standard and provides detailed metadata information as retrieved from the metadata service. The retrieved metadata is also displayed in the AVU format, to allow easy interpretation by the user. A download link is also provided below the output to allow users download the dataset attached to the metdata record if required. This however is an optional feature.

### 5.2.2 VUMC Server Implementation

The server implementation contains all the application code responsible for communication with the metadata services. The VUMC server application code was developed with c++ programming language and contains 5 main classes that are combined to achieve its functionality as shown in figure 5.2.

- Slice API Definition

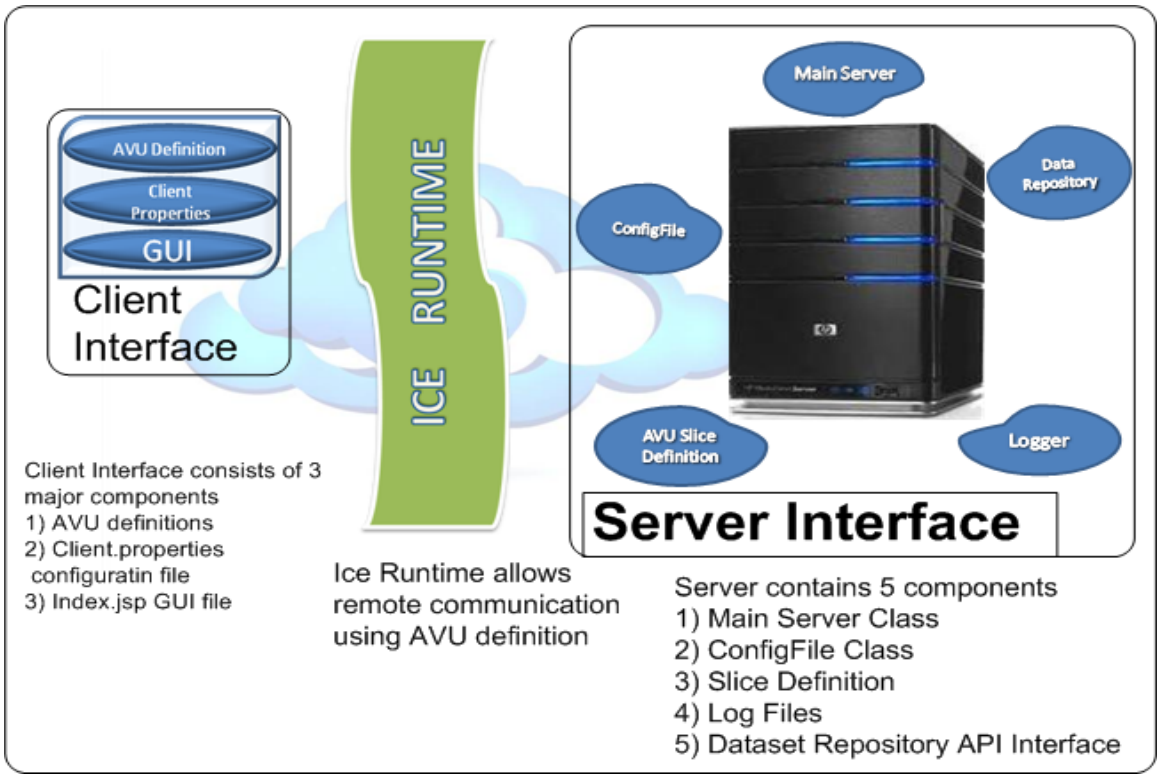


**Figure 5.2:** Client and Server Communication Environment

- Configuration File
- Application code for Metadata service access
- Log Files
- Additional server code

The Slice API definition as earlier pointed out contains all interfaces to be implemented within the server code. For the server, the Slice API definition is compiled using c++ environment functions to generate all the interfaces to be implemented within the application code. Their implementations are combined with the system generated interfaces to allow for execution of necessary operations from them. Since these Slice API interface definitions are system generated, we will not describe them in details within this documentation.

The Configuration file implementation (ConfigFile.cpp) provides methods that maintains and checks for configuration parameters within the VUMC system. It reads through the environ-

ment configuration file and checks for configured server from which it can extract content.

The Log file keeps track of regular connection activities. It stores information such as connection parameters, connection time, error messages.

### 5.2.3 IceGrid Implementation

In ensuring communication amongst all server nodes within the VUMC environment, we adopt the functionalities of an ICE attribute called the IceGrid. IceGrid offers us a powerful collection of integrated features that assist in developing new applications, and also in the deployment and administration of these applications within a distributed dataset environment.

The VUMC server environment is managed using IceGrid registry. For every cluster environment, only one instance of the IceGrid registry will be running. The IceGrid registry acts as a directory for every server node within the system and allows clients create a connection session to the various server environments. The session created can be a normal client session or an administrative session which can either be a secure or non secure session based on the configuration parameters. The registry is configured with the environments server and client endpoint addresses which includes the communication protocol (i.e TCP or UDP), the host IP address and the host port number. Also permissions allowed for communication are specified here and all these information is kept within a master configuration file. The information contained within the IceGrid registry is very vital to the cluster environment as it allows all participating client nodes and server nodes to be discovered, communicate, and share information among themselves. Once all parameters are set for the IceGrid registry, it can be started/activated and communication can now begin within the cluster environment.

For easy connectivity, all participating servers must be configured as nodes within the network to allow easy discovery from the IceGrid registry. The various connected servers are reffered to as IceGrid Nodes. The configuration of the IceGrid Nodes is also dependent on the environment within which the node is running. The communication protocol must match that

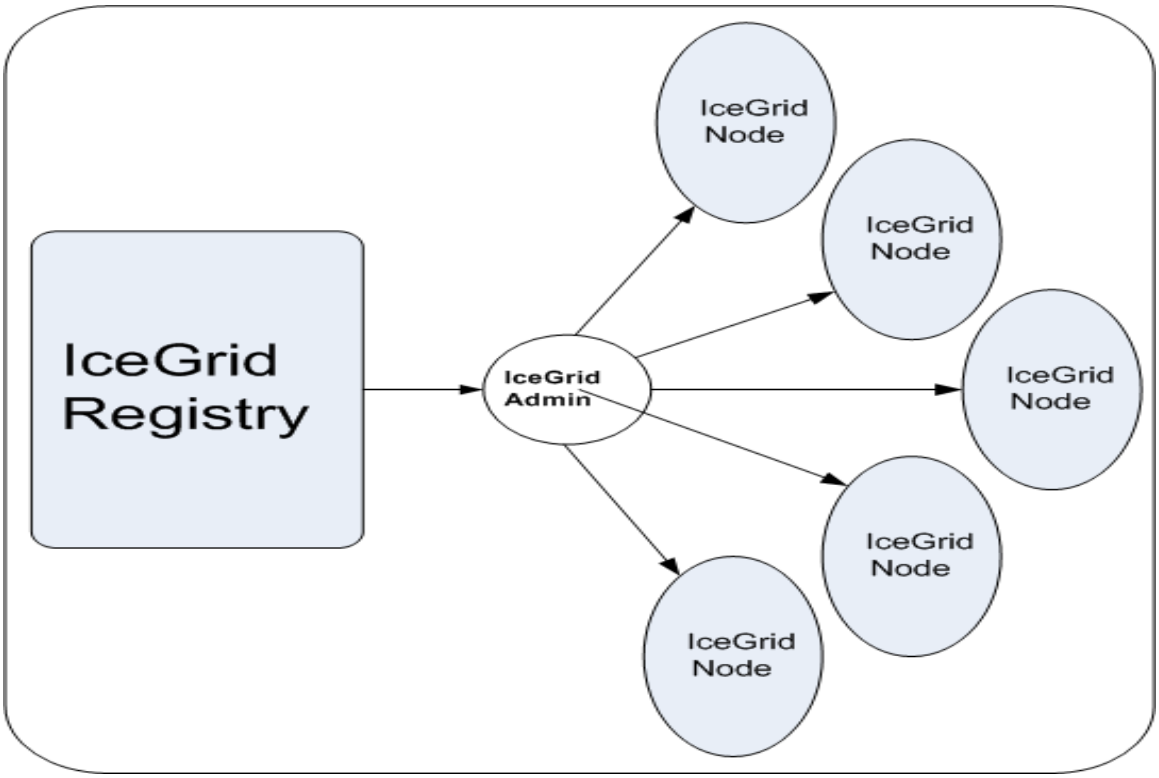


**Figure 5.3:** VUMC IceGrid Environment

running within the IceGrid Registry environment. However, the host IP needs to be modified to match the specific environment and the port number also needs to be specified. An important parameter that must be carefully set is the "IceGrid.Node.Name" parameter. Similarly, the "IceGrid.Node.Data" parameter defines the path to the directory for the specified node. Every node within the cluster must have a unique node name which will distinguish it from other nodes. Also a unique locator proxy parameter needs to be set using the specific IP address of the server environment. After all nodes are carefully configured, they must be activated for the registry to pick them and locate them when required.

Once all the environmental parameters for the IceGrid registry and nodes are set, the IceGrid administrator utility can then be activated 5.3. The IceGrid admin as it is commonly referred to is responsible for total control of the deployed application. It automatically performs activities such as starting a server or modifying a configuration parameter within the environment. For communication with the IceGrid registry, icegridadmin establishes an administrative session to

the registry. From the registry, it can obtain access to all server nodes for communication and managing. Before our application can be deployed we have to provide an XML file that contains the descriptor for the application. The XML file we used for this is the "application.xml" and it contains the application location for the VUMC client and the server. This allows our IceGrid admin access to the application and makes it easy for it to perform operations specified in our application code. After all these parameters are set, we run the IceGrid Admin to register the application information to the icegridnode.

### 5.2.4 VUMC Server Application Implementation

The VUMC server provides a unique framework that can be adopted for achieving communication with multiple metadata services. The major difference in the implementation is with the API's for the different metadata services. The server contains architectural constructs that contain application codes for each metadata service environment. We will now describe the application code and algorithms for communicating with the metadata services and metadata retrieval from them.

The main server implementation can be described with algorithms that detail each unique implementation.

The server implementation is mainly contained in the server.cpp file.

- In the main program we declare 2 variables Status and ic. The status variable contains the exit status of the programmes the ic variable contains the main handle to the ice runtime.
- Next the Ice runtime environment is initialized by calling the Ice::initialize(argc, argv) which returns a smart pointer to the object.
- We then create an object adapter for the object ic which contain the adapter name IcsAdapter, and default port for the server which points to the port location for incoming requests at $cfg.getValueOfKey < string > (\text{"}icsport\text{"})$).
- At this stage a new object of the class TriletsI interface is created to perfom the opera-

tions detailed in the TripletsI function for communicating with the metadata services and performing operations on it. The TripletsI class is explained in details below.

– Now the ic object stays in an active state while all operations are carried out on the other functions and waits for the shutdown signal.

### 5.2.5 TripletsI ()

The TripletsI () provides implementations for the interfaces generated by ICE in main() from the Slice API definition. It contains methods that communicate with the various integrated metadata services to perform required operations. It is within the TripletsI() that we call functions that implement operations that relate to metadata query operations on the various integrated metadata services. For each metadata service, we have a corresponding TripletsI() method that implements the interfaces, thus performing metadata query operations. Each method has a communication interface, a logger interface for keeping up-to-date logs on operations and error checking functionalities for commonly encountered errors. It takes as input the AVU (Attribute, Value & Unit) query parameters. Before performing the metadata query operations, it starts by checking the communication link between the metadata service and the VUMC server environment. If communication can be established, an ok status is printed befor it proceeds to call the methods for performing the operations required. However, if the communication cannot be established an error message is returned to detail the reason for the communication faliure. The log file implemented by the logger interface also stores relevant information on the system status.

The VUMC framework can be extended to accomodate a wide range of metadata services as long as API's can be identified for those metadata services. In subsequent sections, we will describe in details the implementation algorithms for communicating with iRODS, Thredds and MCS just to show how metadata services are integrated with the VUMC system. For the two listed metadata services, the TripletsI() contains the getIrods(), getThredds(), and getMCS() functions that contain relevant application codes for communicating with the iRODS, Thredds and MCS metadata services respectively. FIgure 5.4 shows the relationship between the various

software components of the system. It shows how they connect and operations performed by each component. The getIrods(), getThredds(), and getMCS() functions are described in details and their algorithms are presented next.

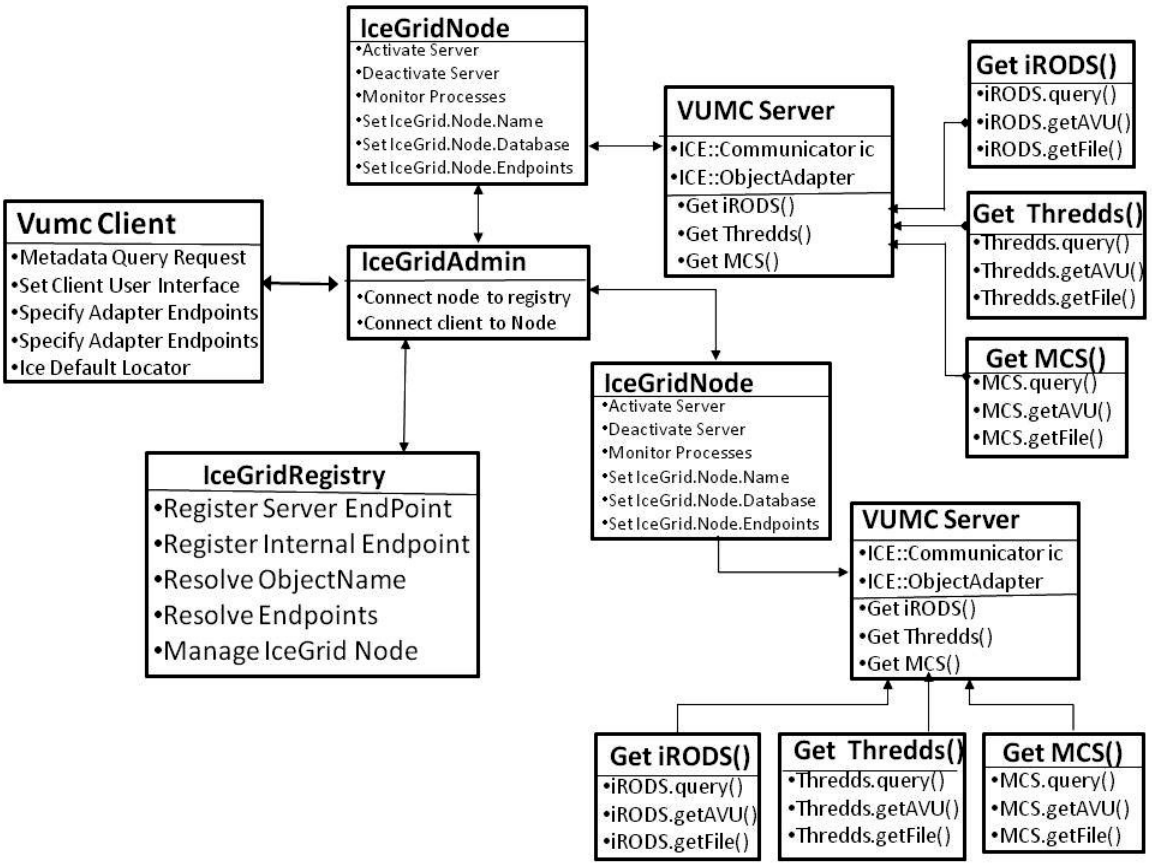


**Figure 5.4:** VUMC Software Implementation Components

### 5.2.6 Metadata Request processing.

Once the AVU is passed using the object adapter to the TripletsI function, the function checks for the functional mappings within the TripletsI class that correspond to the AVU passed. It then passes the AVU to the appropriate class function that corresponds to the metadata query type (i.e getIrods for an iRODS AVU and getThredds for a Thredds AVU ...etc). It is within these methods that the AVU is finally processed accordingly and actual metadata searches are carried out.

For each metadata service of the participating data repositories, we create a "GET" class (i.e. getIrods, getThredds, ...) that make use of the specific API's of the metadata service to perform metadata related operations on the data repository. The Get class performs three main operations from these systems.

1. Metadata Query
2. Metadata Records Retrieval
3. Associated Files Retrieval (optional)

To fully appreciate and understand the implementation we will now describe the algorithms for carrying out searches within the system using our implemented systems iRODS as a reference. Similar functions Thredds are also provided. However we limit the description only to the Thredds and iRODS within this documentation to avoid redundant repetition as the process is very similar for every metadata service integrated. The metadata catalog service (MCS) has also been integrated with the VUMC. We decided to choose iRODS and Thredds for our description due to the significant difference in the architectural implementation of the two metadata services. While iRODS metadata service maintains its metadata within relational database tables using postgres or mySQL, Thredds uses an XML based script to encode and store its metadata records. The method of access for these two metadata catalogs vary significantly and pose very unique challenges in integrating with a system like our VUMC. As a result we are presenting our implementation for integrating these two metadata services to show how the uniqueness of the VUMC for metadata service integration.

### 5.2.7 VUMC Algorithms

The Algorithm 5.1, Algorithm 5.2 and Algorithm 5.3 provide a description of the framework for metadata service integration for the iRODS and Thredds metadata services respectively. The model adopted in the VUMC is basically the same for all integrated metadata services. It takes the metadata query attributes as its input and passes them to functions to perform required operations for metadata query on them. The operations performed are the same across all environments, which are the *query*, $getAVU$ and $getfile$ operations. The *query* operation

searches for the location of the metadata record from within the distributed environment and stores it using a *records* parameter. The *records* becomes the input for the $getAVU$ and $getfile$ operations which is responsible for retrieving all the metadata records within the location and retrieving the actual files respectively. The operations flow is the same across all metadata service environments however the implementation varies based on the platforms of the metadata services and their API structures. The descriptions and implementation of these function are described in details below.

**Algorithm 5.1**: getIrods(*attribute*, *value*, *unit*, *records*, $AVU$,$File$)

**input** : $attribute, value, unit, records$
**output**: $records, AVU, File$

**begin**
  $records \longleftarrow irods.query(attr, val, unit)$
  $AVU \longleftarrow irods.getAVU(records)$
  $File \longleftarrow irods.getfile(records)$
**end**

**Algorithm 5.2**: getThredds(*attribute*, *value*, *unit*, *records*, $AVU$,$File$)

**input** : $attribute, value, unit, records$
**output**: $records, AVU, File$

**begin**
  $records \longleftarrow threddss.query(attr, val, unit)$
  $AVU \longleftarrow thredds.getAVU(records)$
  $File \longleftarrow thredds.getfile(records)$
**end**

**Algorithm 5.3**: getMCS(*attribute*, *value*, *unit*, *records*, $AVU$,$File$)

**input** : $attribute, value, unit, records$
**output**: $records, AVU, File$

**begin**
  $records \longleftarrow mcs.query(attr, val, unit)$
  $AVU \longleftarrow mcs.getAVU(records)$
  $File \longleftarrow mcs.getfile(records)$
**end**

### 5.2.8 IrodsComm.cpp (Algorithm for iRODS environment)

The IrodsComm provides the implementation of the algorithms for the iRODS environment as shown in Algorithm 5.4. The definitions for communicating with iRODS environment are contained within the IrodsComm functions called within the getIrods class. For the iRODS metadata service, we have 3 methods for the 3 main operations to be performed:

1. IrodsComm::query - For metadata query
2. IrodsComm::getAvus - For metadata retrieval and display
3. IrodsComm::getFiles - For retrieving associated files

**Algorithm 5.4**: IrodsComm::Query(*attribute*, *value*, *unit*, *FileList*, *gQueryOut*() )

**input** : $attribute, value, unit$
**output**: $FileList$

```
begin
    r ⟵ nil
    i ⟵ 0
    j ⟵ 0
    k ⟵ 0
    FileList ⟵ 0
    for i ← 0 to gQueryOut() do
        for j ← 0 to gQueryOut() do
            r = gQueryOut().value
            r = gQueryOut().len
            FileList[r]
    return FileList
end
```

#### 5.2.8.1 IrodsComm::Query

For metadata query to be carried out within the iRODS data repository, the input parameters are the attribute, value and unit (AVU), which make up standard metadata format for a file stored within iRODS.

The procedure IrodsComm::Query() takes as input the parameters:

– Attribute: Attribute name of the metadata query

– Value: The corresponding value associated with the attribute

– Unit: The unit of measurement

performs query using

– genQueryOut: an iRODS API function,

and the output generated is stored within a

– FileList

The iRODS metadata service is queried within the IrodsComm::Query() to identify the location of the metadata records that correspond to the AVU's being queried within the metadata registry. It takes in three input for the AVU and uses a minimum of two of the attributes for the query. Using the AVU input as a reference, we call the genQueryOut(), a function within the iRODS API suite for metadata query. The genQueryOut() performs the operation to locate the metadata record and return its location within the data repository using $gQueryOut().value$ and $gQueryOut().len$. The location is stored within $'r'$ as described in the algorithm above. The result which is the location for every matching record, is retrieved and stored within an array in FileList and this result is returned as output. This FileList generated serves as input for the $IrodsComm :: getAVUs$ and $IrodsComm :: getFiles$ functions within the irodsComm class. Within an iRODS metadata service, the record location which we store within FileList links to both the metadata catalog and also the data repository. Obtaining the file location makes it easy to access them using appropriate functions and conditions as maintained within the iRODS environment.

#### 5.2.8.2 IrodsComm::getAVUs

The next algorithm for the IrodsComm:getAVUs() provides the implementation for retrieving all stored metadata content corresponding to the metadata query from the queried metadata service as described in Algorithm 5.5. Using the FileList generated from the $IrodsComm :: Query()$ above as input, the IrodsComm::getAVUs is implemented to retrieve all associated

**Algorithm 5.5**: IrodsComm::getAVUs($FileList$, $genQueryOut$, $Header\ T$)

**input** : $FileList, strbuff$

```
begin
    FileAvus ⟵ avus
    status ⟵= rcGenQuery(_connection, &genQueryInp, &genQueryOut)
    if Status ≠ 0 & status ≠ CAT_NO_ROWS_FOUND then
        error
    else
        TripletList ⟵ triList(genQueryOut− > rowCnt)
        for i ← 0 to genQueryOut− > attriCnt do
            for j ← 0 to genQueryOut− > rowCnt do
                r = genQueryOut− > sqlResult[i].value
                r = j ∗ genQueryOut− > sqlResult[i].len
                Triplet&t = triList[j]
end
```

metadata records associated with the FileList. The process starts by checking the communication link to the iRODS environment. Once communication can be established then the memory location stored within FileList can be accessed to retrieve the metadata records. It recursively checks for all other records that are linked to the FileList locations using the *rcGenQuery*(). The *rcGenQuery*() is a functional API of the iRODS environment that is used specifically for AVU queries with the appropriate conditions. The *genQueryOut*() function of the *rcGenQuery*() class is called to locate the AVU's and retrieve corresponding metadata records associated with it. The result is returned and stored in dynamic database tables within the TripletList() function which correspond to $sqlResult[i].value$ and $sqlResult[i].len$ within the algorithm. It generates a comprehensive list of metadata for the location passed through the FileList function. This result provides additional information on the queried metadata record for ease of referencing by intending users.

#### 5.2.8.3 IrodsComm::getFiles

The final operation in the VUMC metadata reteival process is IrodsComm::getFiles(). After locating the metadata query and associated metadata records within the metadata service us-

**Algorithm 5.6**: IrodsComm::getFiles($FileList$, $strbuff$,$rcDataObjGet$)

**input** : $FileList, strbuff$

```
begin
    it ⟵ string :: iter
    stringobjPath ⟵ _rodsEnv.rodsHome
    dataObjInp_t ⟵ dataObjInp
    for it ← fileList.begin() , it ≠ fileList.end() do
        Status ⟵ rcDataObjGet
        if Status ≠ 0 then
            error
        else
            ifstream ← ifs(locFilePath.c_str())
            strbuff ← ifs.rdbuf()
            it− > second ⟵ strbuff
            status = remove(it− > first.c_str())
end
```

ing the $IrodsComm : query()$ and $IrodsComm : getAVUs()$ respectively, it is important to also identify the actual file associated with the query in order to allow access to it if required. The $IrodsComm :: getFiles()$ procedure provides the functionality for achieving this. As detailed in Algorithm 5.6, it also takes as input the FileList records generated as output from the $IrodsComm :: query()$ function. For every entry in the file list, the filepath is retrieved and passed to function also an iRODS API function. The $rcDataObjGet()$ is used to get a file from the iRODS environment. At this stage the $rcDataObjGet()$ function is executed to retrieve the data object stored within the FilePath. These data objects corresponds to files within the iRODS data repositories and these result(data objects) are stored in a status string. This is an iterative process that continues until the last object within the FileList is checked. The output is then passed back to the getIrods() function on the server side as a result for the query. The server in turn passes them to the client and provides a download link for access and retrieval of the actual file from the data repository. These three outlined algorithms summarize the complete process of metadata query from an existing metadata service using the VUMC system.

While the processes described above is for the iRODS metadata service, we have adopted the system to also query the Thredds metadata service and the Metadata Catalog Service (MCS). As shown in Algorithm 5.2 and Algorithm 5.3 the procedure followed in carring out metadata queries is similar to that used with iRODS. The operations performed are basically the same i.e

1. Locating metadata records within the metadata service (Metadata Query)
2. Retrieval of associated metadata records (AVU Retrievals)
3. Associated datasets retrieval

This outlined framework provides a unique architecture that can be easily extended to integrate and achieve communication with other metadata services. Its direct and well defined process flow allows metadata query and retrieval using easily adaptable and non-complex operations. Currently, the VUMC provides support for three metadata services which are iRODS, Thredds and Metadata Catalog Service (MCS). We strongly believe that with the successful integration of a wide range of metadata services, metadata sharing will pose less challenges in the future and collaborative research initiatives can be pursued more effectively.

### 5.2.9 ThreddsComm.cpp(Algorithm for Thredds environment)

We adopt the same architecture and used similar procedures for the Thredds environment to carry out metadata query. Just as described with iRODS, the operations performed are:

- ThreddsComm::query - For metadata records identification
- ThreddsComm::getAvus - For associated metadata retrieval and display
- ThreddsComm::getFiles - For retrieving associated files

The major distinguishing factor from iRODS implementations is with the API's called for performing the required operations on the metadata services. Thredds metadata is stored within catalogs that are embedded within XML tags. We therefore employed XML parsing algorithms to identify the metadata paths and retrieve the metadata records from the Thredds catalog.

In similar fashion, the associated metadata records linked to the file path are retrieved and displayed to the user through the client interface. Actual data file are then retrieved and a download link is provided for the via the VUMC client for the user to access the actual files if needed.

The outputs of these methods described in the algorithms below is passed to the getThredds function.

#### 5.2.9.1 ThreddsComm::query()

**Algorithm 5.7**: ThreddsComm::query(*attribute*, *value*, *unit*, *processCatalog*, *records*() )
**input** : $attribute, value, unit$
**output**: $recordSet$
**begin**
  $records \longleftarrow processCatalog(attr, val, unit)$
**end**

This function as described in Algorithm 5.7 takes the AVU as input and does a query to locate the record set within the TDS metadata catalogue. This operation is performed with just a single query process. Upon receiving the AVU's as inputs it passes them into a function processCatalog as described in Algorithm (processCatalog) below. This processCatalog function checks within the Thredds Data server (TDS) XML catalogue iteratively for matching records and returns them to the $ThreddsComm :: query()$ where it is returned as $avuFilesContent$. The location kept within records is then passed for further queries for additional metadata and the actual data file.

#### 5.2.9.2 ThreddsComm::ProcessCatalog()

The ProcessCatalog() function in Algorithm 5.8 is an XML parsing algorithm for scanning through the XML metadata file for required records. The process starts by checking through the TDS directories for all available metadata catalogs (metaRecs) using the $getMetaRecord()$. Once the metadata records are identified, it then scans through each catalog node to locate the

dataset paths stored within them. This provides a point of reference to the metadata records stored within the TDS. It is important to note that the datasets cannot be distinguished from each other until the metadata records are processed to identify each available dataset from the other. Once the dataset directory is located within the catalogues, the processCatalog() function can now locate the metadata records using the file path. ProcessCatalog() function gets all metadata records within the catalog and checks them by looping through to find matching records to the AVU entered as input. Once it finds the match it pushes it into a set of strings $(attr, val, unit)$ from which it can be retrieved.

**Algorithm 5.8**: ThreddsComm::processCatalog(*attribute*, *value*, *unit*, *urlset*, *avupath*() )

**input** : $attribute, value, unit$
**output**: $recordSet$

```
begin
    if tag == "catalog" then
        metaRecs ⟵ getMetaRecord
        while (metaRecs ≠ endofFile) do
            if tag == "dataset" then
                mnode ⟵ getMetaNode()
                while (mnode ≠ endofFile) do
                    if tag == "metadata" then
                        attr ⟵ getAttr()
                        val ⟵ getVal()
                        unit ⟵ getUnit()
                        if (attr ≠ nullANDattr == a; value ≠ nullANDval == v; unit ≠
                        nullANDunit == u) then
                            avuPnode ⟵ getRecord()
                            if avupath ≠ null then
                                recordSet.insert(avupath)
    return FileList
end
```

#### 5.2.9.3 ThreddsComm::getAvus

Once the metadata records has been located using the above listed algorithms, the full metadata list then can be accessed using the $ThreddsComm :: getAvus$ function. The $ThreddsComm :: getAvus$ function performs a similar operation to the processCatalog() function. However rather than just retrieving the metadata paths, it iteratively scans through the catalog and retrieves all the AVU metadata records associated with datasets linked to the path. It takes as input the record path retrieved from the $threddsComm :: Query()$ function. When this path address is identified, it then scans through the catalog referenced by the path for all associated metadata records and returns this as the output.

#### 5.2.9.4 ThreddsComm::getFiles

In the same way, the $ThreddsComm :: getAvus$ locates the files associated with the metadata records within the thredds environment and returns the Filepath and FileName. This provides access to the file and allows for download where necessary.

### 5.2.10 mcsComm.cpp(Algorithm for MCS environment)

This same process flow is implemented for the metadata catalog service MCS and similarly for other metadata services to be integrated within the VUMC system. The MCS however provides a collection, view and item management operational interface. We therefore make use of the item operation API's to query for individual metadata. This allows metadata to be queried in similar fashion to the iRODS and Thredds and thus a single VUMC client implementation can query all three metadata services simultaneously.

Using the globus toolkit environment, the MCS API functions $ListAttributes()$, $Query()$, $QueryCollection()$, $QueryFile()$, $QueryView()$ were integrated to achieve metadata query and File recovery from thr MCS environment. The entire process is the same from iRODS and

Thredds and as a result we will not go into the algorithms.

### 5.2.11 VUMC Cache Implementation Algorithm

For cache update as detailed in Algorithm 5.9, the probabilistic ratio is first checked. If it returns false then the record cannot be cached and the process terminates, otherwise the parameter list and AVU records are stored within newAVU for caching. The cache is subsequently updated with the records within the newAVU. The cached information include the object adapter information and proxy information. While updating the cache, the cache threshold is also constantly monitored. Once the threshold is full it uses the *getleastused*() to identify the least recently used and evacuates from within the cache to create space for new records to be stored.

**Algorithm 5.9**: LocalCacheUpdate($params$, $files$, $avus$)

**input** : $params, files, avus$
**output**: $avus, files$

```
begin
    while (paramlist ≠ null) do
        if probabilityChk = false then
            terminate
        else
            newavu ← partialRemotecopy(paralist, avu)
            localcache.update(newavu, files)
            if localcache.size > maxCacheSize then
                for (i ← fractiontoremove(); i > 0; i − −) do
                    k ⟵ localcache.getleastused()
                    localcache.remove(k)
end
```

Algorithm 5.10 details the operations within the cache query process. This process queries the cache for AVU parameters and returns corresponding triplet list as its output. It takes as input the AVU parameters and file name. As described in the algorithm, when an AVU query is executed the system checks the local cache for matching AVU triplets. If the query returns a null value, it signifies that no matching query was found and execution is terminated with no further operations performed on the cache. However, if the query result is not null, the

**Algorithm 5.10**: IfGetAVUs(*params*, *files*, *avus* )

**input** : $params, files, avus$
**output**: *tripletList*

```
begin
    triplets ⟵ cacheset(params)
    if triplets ≠ null & triplets ≠ localget.Triplets(triplets, param, file, avu) then
        return tripletList
    else
        triplet ⟵ RemotegetTriplet(triplets, param, file, avu)
        if triplet ≠ null) then
            RemoteCacheUpdate(triplets, param, file, avu)
    return tripletList
end
```

RemoteGetTriplet() method is executed to obtain the triplet list from its remote location. Finally, a RemoteCacheUpdate() method is executed to update the cache with the new metadata AVU records. These functions are more clearly explained below in details.

**Algorithm 5.11**: RemoteGetTriplets(*params*, *files*, *avus*)

**input** : $params, files, avus$
**output**: *tripletList*

```
begin
    proxiesList ← query.FindAllProxy(avu)
    if proxiesList == null || Proxy.Lenght ≤ 0 then
        return null
    else
        for (i = 0; i < proxiesList.length; i++) do
            triplet ⟵ GetTriplets(proxyList.object)
            if (triplet ≠ null) then
                tripletList.Add(triplet)
        return tripletList
end
```

The RemoteGetTriplet() function detailed in algorithm 5.11 takes the same queried AVU parameter or file list as input, and returns as the output the tripletList from remote server locations. The standard proxiesList contained within the location service is required for completing this process. The proxiesList provides a look up table for identification of remote servers

within the network.The query.FindAllProxy() function is called to identify the proxy addresses required from within the proxiesList. Once all records within the proxiesList is queried and the output is null then the operation is terminated as the proxy address is not available within the list, meaning the server does not belong to the network. However, if the condition is satisfied and the outcome is not null, the GetTriplets() method is executed on the identified proxy. If matching entries are found the are compiled into a tripletList which is returned as the process output.

# Chapter 6

# Experimental Performance Evaluations

We perform some experiments to evaluate and compare the system performance for the VUMC before and after cache implementation. In the first sets of experiments, we checked the hit ratio with respect to frequency of cache hits relative to increase in cache sizes. The queries where carried out using the LRU and LFU algorithm implementations to compare and check their performances. This experiment was conducted to test when query is conducted only to iRODS servers, Thredds servers and any of the two servers simultaneously. We measure the performance ratings for these conditions and explain the trends.

## 6.1 Experimental Environment

We implemented the VUMC in C++ and tested on three(3) server environments. The first server environment running the ICEGrid Registry was compiled and built on GNU g++ compiler on a 64-bit intel i7 multi-core processor running on Ubuntu Linux 12.04 LTS operating system. This is the main server machine that runs the VUMC for remote communication using the IceGridRegistry which connects all remote servers for communication within the environment. We also built our client on this machine to establish remote communication to remote machine running the metadata services.

The second environment contains the first server node which runs the iRODS metadata service built over a postgres database containing metadata records for iRODS. The metadata records

are organized in different categories in key-value pairs. The system is a 64-bit Intel®Core quad-core i5-2500 CPU with 3.30GHz per core CPU capacity processor running on an Ubuntu Linux 11.10 operating system.

The Thredds server node was built on the third machine with its metadata service residing here. This machine runs on ubuntu 12.04 LTS with a 64-bit iPentium(R) Dual-Core CPU T4300 running on 4GB of RAM at 2.10GHz CPU capacity. Our thredds metadata server contains various test records with its metadata stored in several xml metadata files within the metadata service. Since Thredds accounts for a small proportion of the system requirements, the MCS is also developed on this system. The MCS is compiled over a MySQL relational database environment which contains the metadata records.

For all the experiments performed, a sample size of 1 million metadata records residing on remote metadata servers was used to test the systems performance. These records consist an even mix of iRODS metadata, Thredds metadata records and MCS metadata files in HDF and NetCDF formats.

## 6.2 Data Sources

For the iRODS environment, we made use of HDF5 datasets generated for scientific data. The Thredds environment contains a generated set of NetCDF datasets which best suit the environment. We decided to seperate these datasetsamongst the two metadata services (i.e HDF for iRODS and NetCDF for Thredds) to easily distinguish when records are retrieved.

## 6.3 Performance Evaluation

### 6.3.1 Cache Performance

We conducted three sets of experiments to compare system performance for the VUMC before and after cache implementation. In the first sets of experiments, we evaluated the changes in hit ratio with respect to frequency of cache hits for an increasing set of records. We also conducted experiments to evaluate the effects of implementing a cache on the query time as some delay is expected. It compares the time differences for remote query and checks to see if the latency is improved. We measure the performance ratings for these conditions and explain the trends.

The results showed significant improvements in both the hit rate and query time. After the cache was implemented, more hits were experienced in all cases. A steady improvement in fine proportions was observered. For the query time, as the processed records increased, the performance largely improved. This can be attributed to the fact that the cache eliminates ambiquous remote searches and streamlines the search to specific areas where the queried records can be found.

### 6.3.2 Delay and Hit Rate Evaluation

The second set of experiment conducted was to check the delay experienced in querying the cache as the cache size increases. We evaluated the delay probabilities using 3 different average delay values. The result, figure 6.3 is a fine gradient curve which shows some good measure of performance. As the probability ratio decreases, the delay experienced also decreases significantly.

We also evaluate the probability of a cache hit for a query made. We check by using 3 values for average probability that a record will be accessed. Also in this case figure 6.4, as the cache size increases and the probability ratio increases, the hit ratio also increases as expected.

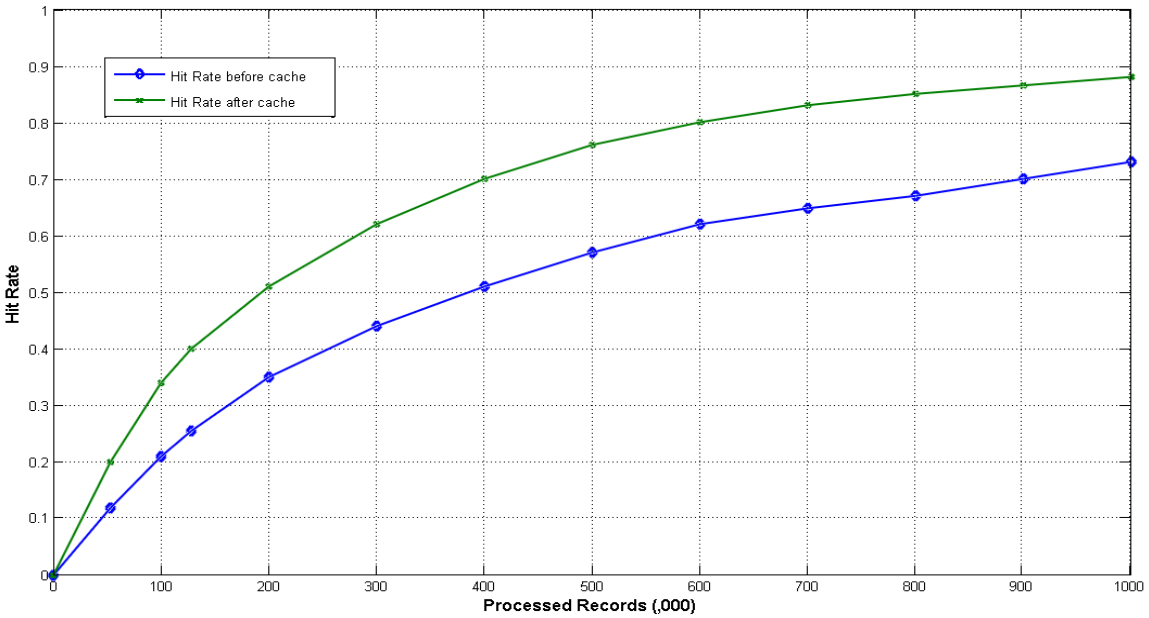


**Figure 6.1:** Comparism of Hit Rate before and after cache implementation

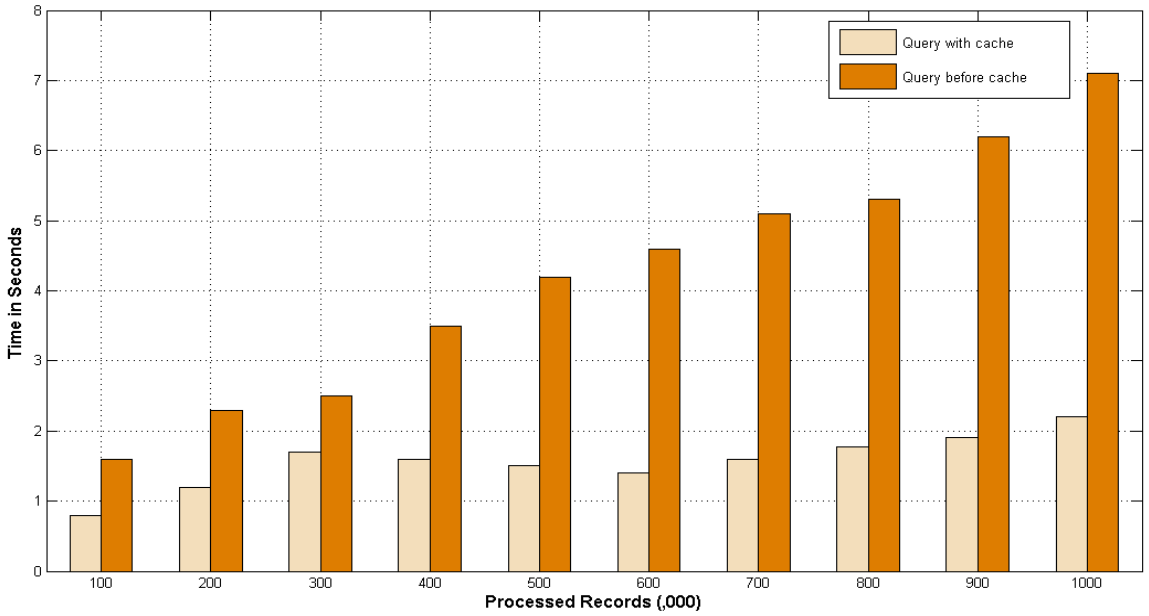


**Figure 6.2:** Comparism of Query Time before and after cache implementation

### 6.3.3 Cache Size Influence

Finally we conducted experiments to evaluate the effect of changes in cache size on the main performance measures, which are delay experienced and the hit rate. The graphs shown in Figure 6.5 and Figure 6.6 respectively show that as the cache sizes increases the effect on tbe

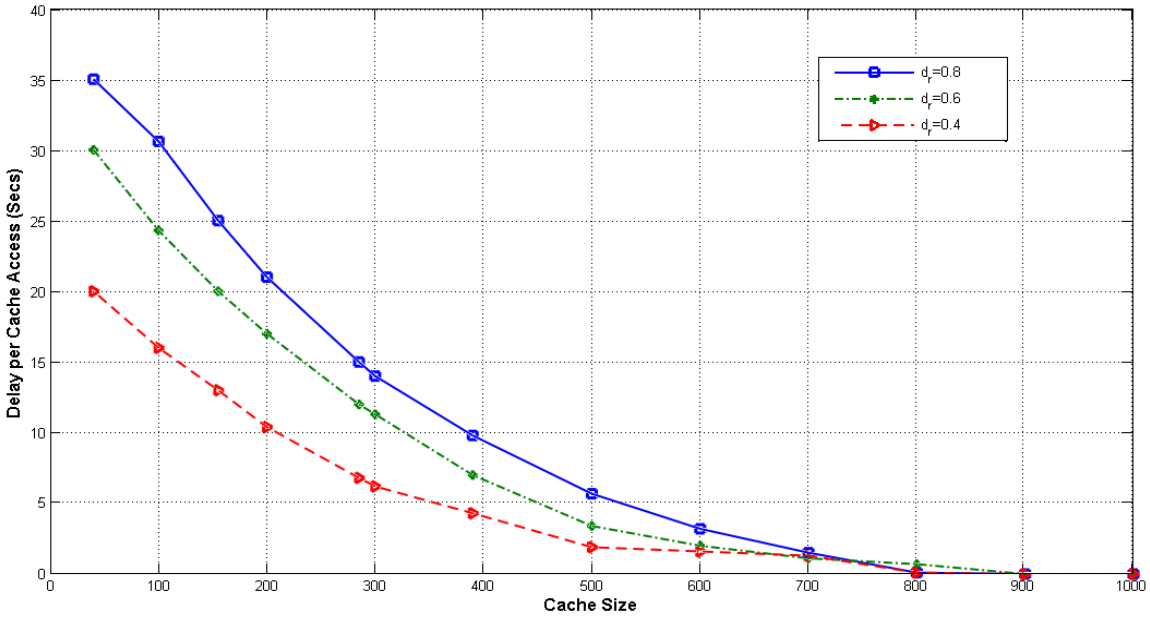


**Figure 6.3:** Probability of Delay in Cache

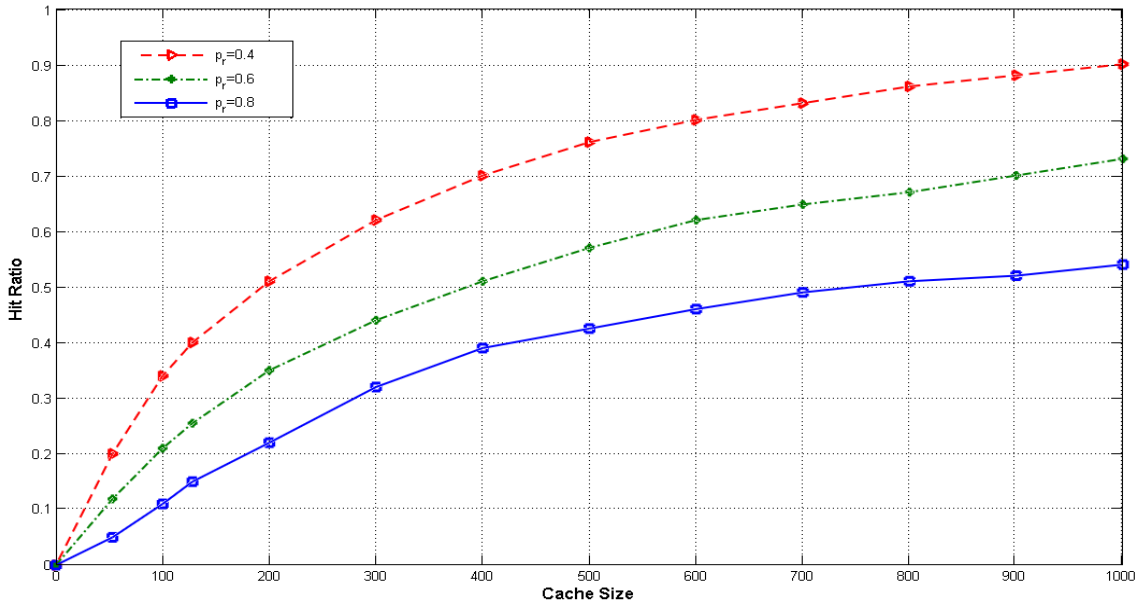


**Figure 6.4:** Cache Hit Probability

hit ratio and delay are inverse. An increase in the cache size results in an improvement in hit ratio, and also results in a lower delay. This estimations suggest improved performance with better utilization of the cache.

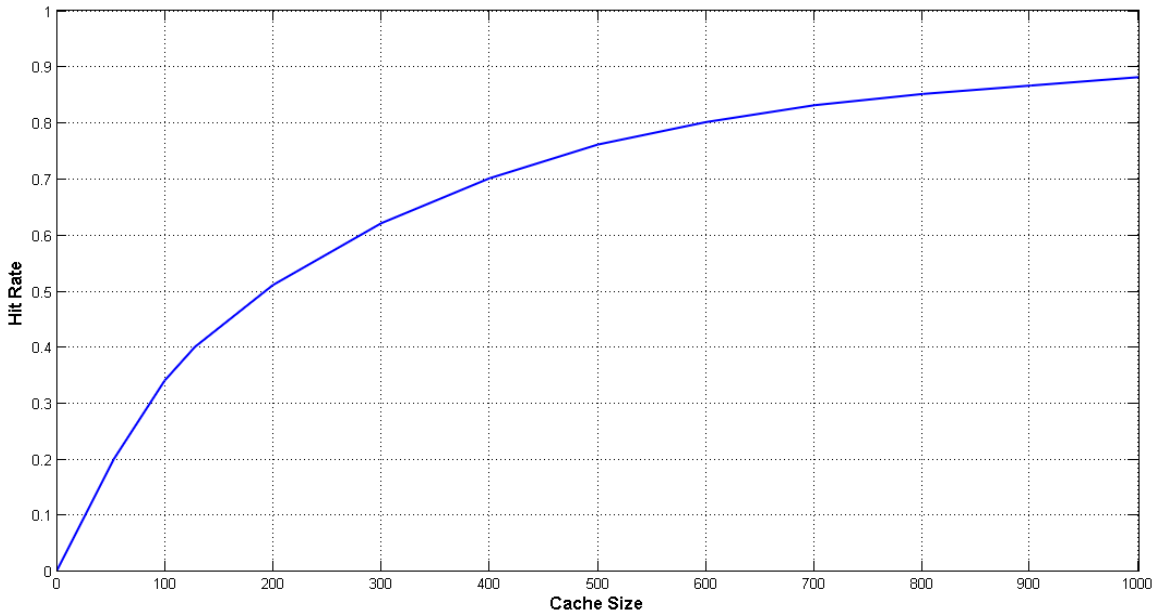


**Figure 6.5:** Effect of Cache Size on Hit Rate

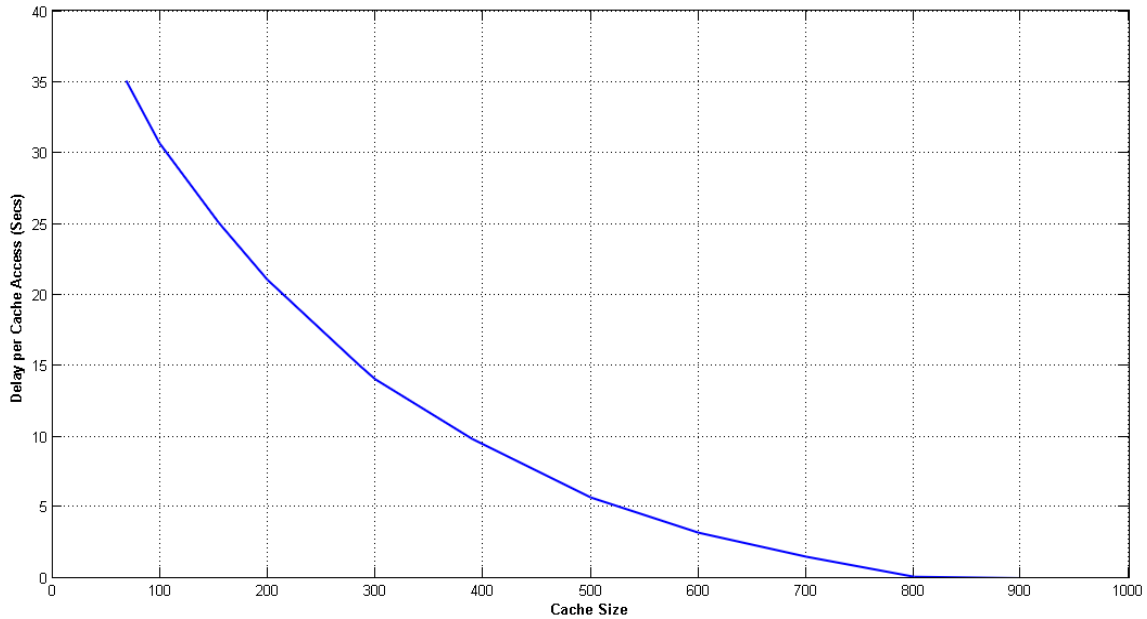


**Figure 6.6:** Effect of Cache Size on Delay

# Chapter 7

# Summary

We have studied and discussed datasets communication within a distributed repository environment. We have also examined the importance of metadata in dataset discovery over such environments and the role of the metadata service in achieving this. We have also examined challenges with communication among various metadata services over distributed environments. We introduced the virtual universal metadata catalog(VUMC) for integrating existing metadata services and allowing communication between them. We described its framework and modalities for integrating metadata service. Its implementation is independent of the various metadata services and any metadata service can be integrated using their distinct API's for communication, query, and data transfer. We detailed its implementation and demonstrated it functionalities using two existing metadata services iRODS, THREDDS. We showed how it can be implemented over other metadata services by providing algorithms that demonstrated how the systems were integrated. The API's called and their description is determined by the metadata service environment and this largely influences the implementation for each server node environment. Also we described a caching implementation for the VUMC using a probability driven cache design. For each environment, a time factor comes into play in identifying records to be cached. The frequently queried records are stored within the cache within the threshold limit. On exhaustion of the cache threshold, we implemented the LRU caching policy for metadata replacement. This allows faster queries within the VUMC environment and improves performance significantly. We also showed with our experimental results the improvement in performance with the implementation of the caching architecture and its importance for efficient query process within the VUMC.

## 7.1 Conclusion

In general, the VUMC provides a unique architecture for metadata sharing within distributed environments. Moving away from how data access is conducted at the moment, we believe the VUMC can achieve fast and accurate access to datasets. By querying for metadata, the query time is significantly reduced and this allows for more efficient query processing. The cache also reduces query time as deterministic query can be conducted based on the entries stored within the cache. The performance evaluation shows an overall good performance and thus encourages improvement on what has been done to achieve an improved system.

## 7.2 Future work

In the future we intend to create an improved mechanism for distributed access of metadata, that can intelligently predict dataset locations based on previous metadata queries so that access time is minimized and query processing is more efficiently achieved. Also we will optimize the caching process to monitor and reduce any additional overhead cost experienced as a result of cache searches.

# References


Ahmed, M. U., Zaheer, R. A. & Qadir, M. A. (2005), Intelligent cache management for data grid, *in* 'Proceedings of the 2005 Australasian workshop on Grid computing and e-research - Volume 44', ACSW Frontiers '05, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, pp. 5–12.
**URL:** *http://dl.acm.org/citation.cfm?id=1082290.1082292*

Allcock, B., Chervenak, A., Foster, I., Kesselman, C. & Livny, M. (2005), 'Data grid tools: enabling science on big distributed data', *Journal of Physics: Conference Series* **16**(1), 571.

Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B. & Mock, S. (2004), Kepler: an extensible system for design and execution of scientific workflows, *in* 'Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on', pp. 423–424.

Apps, A. (2005), *Guidelines for Encoding Bibliographic Citation Information in Dublin Core Metadata*.

Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R. & Ives, Z. (2007), Dbpedia: A nucleus for a web of open data, *in* K. Aberer, K.-S. Choi, N. Noy, D. Allemang, K.-I. Lee, L. Nixon, J. Golbeck, P. Mika, D. Maynard, R. Mizoguchi, G. Schreiber & P. Cudré-Mauroux, eds, 'The Semantic Web', Vol. 4825 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 722–735.
**URL:** *http://dx.doi.org/10.1007/978-3-540-76298-0*$_5$2

berkeley.edu (2012), 'Boinc:open-source software for volunteer computing and grid computing.', `http://http://boinc.berkeley.edu//`.

Boose, E. R., Ellisona, A. M., Osterweil, L. J., Clarke, L. A., Podorozhny, R., Hadley, J. L., Wise, A. & Foster, D. R. (2007), 'Ensuring reliable datasets for environmental models and forecasts', *Ecological Informatics* **2**(3, Sp. Iss. SI), –247.

Bruce E. Bargmeyer, Gillman, D. W. (2000), Metadata standards and metadata registries: An overview, *in* 'International Conference on Establishment Surveys II'.

Cathro, W. (1997), Metadata An overview Paper given by Dr. Warwick Cathro, Assistant Director-General, Services to Libraries Division at the Standards Australia Seminar, "Matching Discovery and Recovery" , Technical report.

Chang, C., Kurc, T., Sussman, A. & Saltz, J. (2000), 'Optimizing retrieval and processing of multi-dimensional scientific datasets'.

Chervenak, A., Foster, I., Kesselman, C., Salisbury, C. & Tuecke, S. (1999), 'The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets', *JOURNAL OF NETWORK AND COMPUTER APPLICATIONS* **23**, 187–200.

Cornillon, P., Gallagher, J. & Sgouros, T. (2003), 'Opendap: Accessing data in a distributed, heterogeneous environment', *Data Science Journal* **2**, 164–174.

Deelman, E., Singh, G., Atkinson, M., Chervenak, A., Chue Hong, N., Kesselman, C., Patil, S., Pearlman, L. & Su, M.-H. (2004), Grid-based metadata services, *in* 'Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on', pp. 393–402.

DICE (2009), FACT SHEET: iRODS integrated Rule Oriented Data System Open Source Data Grid, Helping People Organize and Manage Large Collections of Distributed Digital Data, Technical report, University of North Carolina at Chapel Hill.

Dursi, J. (2012), 'Formats for scientific data management netcdf4, and hdf5', *http://wiki.scinethpc.ca/wiki/images/a/af/Netcdfhdf5.pdf* .

ESRI (2002), 'Metadata and gis: An esri® white paper', *http://www.esri.com/library/whitepapers/pdfs/metadata-and-gis.pdf* .

Evron, S. (2009), A practical guide to data caching with zend server, White paper, Zend Technologies, Inc, Zend Technologies, Inc. 19200 Stevens Creek Blvd. Cupertino, CA 95014, USA.
**URL:** *http://static.zend.com/topics/Zend-Server-Data-Caching-Whitepaper-0106-T-WP-R1-EN.pdf*

Feng, G., Yeung, K., Wong, H.-L. & Kheong, S. C. (2000), Optimal cache allocation and probabilistic caching for local loss recovery in reliable multicast, *in* 'Communications, 2000. ICC 2000. 2000 IEEE International Conference on', Vol. 3, pp. 1401–1405 vol.3.

FGDC (2012), 'Federal geographic data committee', `http://www.fgdc.gov/metadata/index_html`.

Fiolek, A. (2008), National oceanographic data center publications and products in the noaa central library network, 1961-present, *in* 'Library and Information Services Division Cur-

rent References 2008-2', National Oceanographic Data Center, National Oceanic and Atmospheric Administration.

Fletcher, I. (2013), 'Federated search: The options', `http://www.searchtechnologies.com/federated-search.html`.

Gehani, A., Kim, M. & Malik, T. (2010), Efficient querying of distributed provenance stores, *in* 'Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing', HPDC '10, ACM, New York, NY, USA, pp. 613–621.
**URL:** *http://doi.acm.org/10.1145/1851476.1851567*

globus.org (2012), 'Globus:open source grid software for distributed resource sharing', `http://http://dev.globus.org/wiki/Welcome/`.

Henning, M., Spruiell, M., Boone, D., Eagles, B., Laukien, M., Fouc, B., Normier, B., Newhook, M. et al. (2009), *Distributed Programming with Ice*, www.zeroc.com/Ice-Manual.pdf,.

Jelenković, P. R. & Radovanović, A. (2004), 'Optimizing lru caching for variable document sizes', *Comb. Probab. Comput.* **13**(4-5), 627–643.
**URL:** *http://dx.doi.org/10.1017/S096354830400625X*

Johnston, S., Fangohr, H. & Cox, S. (2008), 'Managing Large Volumes of Distributed Scientific Data', *Computational Science – ICCS 2008* (5103), 339–348.

Kansa, S. W., Kansa, E. C. & Schultz, J. M. (2007), 'An open context for "near eastern archaeology"', *Near Eastern Archaeology* **70**(4), pp. 188–194.
**URL:** *http://www.jstor.org/stable/20361331*

Law, D. (2013), 'Arcgis for server 101; understanding architecture, deployment, and workflows', *http://www.esri.com/esri-news/arcuser/spring-2013/ /media/Files/Pdfs/news/arcuser/0313/arcgis101.pdf* pp. 42–47.

Leonard, W. & Albee, P. B. (2010), Cross engine database joining., *in* O. Ormandjieva, C. Constantinides, A. Abran & R. Y. Lee, eds, 'SERA', IEEE Computer Society, pp. 19–26.

Liu, L., He, G., Shi, X. & Song, H. (2007), Metadata extraction based on mutual information in digital libraries, *in* 'Information Technologies and Applications in Education, 2007. ISITAE '07. First IEEE International Symposium on', pp. 209–212.

Manikantan, R., Rajan, K. & Govindarajan, R. (2012), 'Probabilistic shared cache management (prism)', *SIGARCH Comput. Archit. News* **40**(3), 428–439.
**URL:** *http://doi.acm.org/10.1145/2366231.2337208*

McClelland, M. (2003), 'Metadata standards for educational resources', *Computer* **36**(11), 107–109.

Megiddo, N. & Modha, D. S. (2003), 'One up on lru', *THE MAGAZINE OF USENIX SAGE August 2003* **28**(4), 7–11.

Michael Witt, M. G. (2012), Datalib, *in* '7th International Conference on Open Repositories in Edinburgh, Scotland.', Purdue University, Pennsylvania State University - Main Campus, American Library Association Annual Conference in Anaheim, California,.

Microsystems, S. (2008), 'Designing and implementing scalable applications with memcached and mysql', *A MySQL® White Paper* **1**.
**URL:** *http://mahmudahsan.files.wordpress.com/2009/02/mysql$_w$p$_m$emcached.pdf*

milkyway.cs.rpi.edu (2012), 'Milkyway@home', `http://milkyway.cs.rpi.edu/milkyway/`.

Milstead, J. & Feldman, S. (1999), 'Metadata: Cataloging by Any Other Name.', *Online* **23**(1).

Mookerjee, V. S. & Tan, Y. (2002), 'Analysis of a least recently used cache management policy for web browsers', *Operations Research* **50**(2), 345–357.
**URL:** *http://pubsonline.informs.org/doi/abs/10.1287/opre.50.2.345.430*

MSGIC (1998), 'Msgic tools for effective gis management', `http://www.msgic.state.md.us/publicat/gismngmt/part1/tsld037.htm`.

Ordille, J. & Miller, B. (1993), Distributed active catalogs and meta-data caching in descriptive name services, *in* 'Distributed Computing Systems, 1993., Proceedings the 13th International Conference on', pp. 120–129.

Pandya, R., Domenico, B. & Marlino, M. (2003), Finding and using data in educational digital libraries, *in* 'Proceedings of the 3rd ACM/IEEE-CS joint conference on Digital libraries', JCDL '03, IEEE Computer Society, Washington, DC, USA, pp. 399–399.
**URL:** *http://dl.acm.org/citation.cfm?id=827140.827230*

Press, N. (2004), *Understanding Metadata*, National Information Standards Organization Press.

Psaras, I., Chai, W. K. & Pavlou, G. (2012), Probabilistic in-network caching for information-centric networks, *in* 'Proceedings of the Second Edition of the ICN Workshop on Information-centric Networking', ICN '12, ACM, New York, NY, USA, pp. 55–60.
**URL:** *http://doi.acm.org/10.1145/2342488.2342501*

Quinones, E., Berger, E., Bernat, G. & Cazorla, F. (2009), Using randomized caches in probabilistic real-time systems, *in* 'Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on', pp. 129–138.

Reiner, B. & Hahn, K. (2004), 'Optimized management of large-scale data sets stored on tertiary storage systems', *IEEE Distributed Systems Online* **5**.

Santos, N. & Koblitz, B. (2006), 'Metadata services on the grid', *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* **559**(1), 53 – 56. ¡ce:title¿Proceedings of the X International Workshop on Advanced Computing and Analysis Techniques in Physics Research¡/ce:title¿ ¡ce:subtitle¿ACAT 05¡/ce:subtitle¿ ¡xocs:full-name¿X International Workshop on Advanced Computing and Analysis Techniques¡/xocs:full-name¿.
**URL:** *http://www.sciencedirect.com/science/article/pii/S0168900205022308*

Schwarte, A., Haase, P., Hose, K., Schenkel, R. & Schmidt, M. (2011), Fedx: optimization techniques for federated query processing on linked data, *in* 'Proceedings of the 10th international conference on The semantic web - Volume Part I', ISWC'11, Springer-Verlag, Berlin, Heidelberg, pp. 601–616.
**URL:** *http://dl.acm.org/citation.cfm?id=2063016.2063055*

Seguin, K. (2012), 'The little redis book', **1**.
**URL:** *http://openmymind.net/redis.pdf*

Simbad (2013), Simbad astronomical database, *in* 'SIMBAD Website', SIMBAD database, CDS, Strasbourg, France.
**URL:** *http://simbad.u-strasbg.fr/Simbad*

Singh, G., Bharathi, S., Chervenak, A., Deelman, E., Kesselman, C., Manohar, M., Patil, S. & Pearlman, L. (2003), A metadata catalog service for data intensive applications, *in* 'Proceedings of the 2003 ACM/IEEE conference on Supercomputing', SC '03, ACM, New York, NY, USA, pp. 33–.
**URL:** *http://doi.acm.org/10.1145/1048935.1050184*

Slater, M. & Chrysanthou, Y. (1997), View volume culling using a probabilistic caching scheme, *in* 'Department of Computer Science, University College London', ACM Press, pp. 71–78.

Swan, B. (2011), 'Performance tuning php apps on windows with wincache', *Microsoft Web* **1**.
**URL:** *http://www.microsoft.com/web/post/performance-tuning-php-apps-on-windows-with-wincache*

Thaler, D. G. & Ravishankar, C. V. (1998), 'Using name-based mappings to increase hit rates', *IEEE/ACM Trans. Netw.* **6**(1), 1–14.
**URL:** *http://dx.doi.org/10.1109/90.663936*

Tim, Mark, D., Valorie, H. & Robin, T. D. (n.d.), 'Official DSpace Documentation'. https://wiki.duraspace.org/display/DSPACE/Home.

Tim Robertson, Doring, M., Wieczorek, J. & Renato De, Giovanni, D. V. (n.d.), *http://rs.tdwg.org/dwc/terms/guides/text/index.htm"*, *month* = .

UNIDATA (n.d.), 'THREDDS Fact Sheet - The THREDDS Data Server', *http://www.unidata.ucar.edu/projects/THREDDS/* .

Vakali, A. (2000), Lru-based algorithms for web cache replacement, *in* 'Proceedings of the First International Conference on Electronic Commerce and Web logies', EC-WEB '00, Springer-Verlag, London, UK, UK, pp. 409–418.
**URL:** *http://dl.acm.org/citation.cfm?id=646160.680189*

Ward, J. H., Torcy, A. d., Chua, M. & Crabtree, J. (2009), Extracting and ingesting ddi metadata and digital objects from a data archive into the irods extension of the nara tpap using the oai-pmh, *in* 'Proceedings of the 2009 Fifth IEEE International Conference on e-Science', E-SCIENCE '09, IEEE Computer Society, Washington, DC, USA, pp. 185–192.
**URL:** *http://dx.doi.org/10.1109/e-Science.2009.34*

Wein, D. (2002), *Understanding The Metadata Cache.*

ZeroC, I. (2013), 'Welcome to zeroc, the home of ice website', *http://zeroc.com/index.html* .

Zhang, S., Coddington, P. & Wendelborn, A. (2009), Davis: A generic interface for irods and srb, *in* 'Grid Computing, 2009 10th IEEE/ACM International Conference on', pp. 74–80.