3-28-2007

A grid computing network platform for enhanced data management and visualization

Javier Delgado
Florida International University

DOI: 10.25148/etd.FI14062236

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

A GRID COMPUTING NETWORK PLATFORM FOR ENHANCED DATA MANAGEMENT AND VISUALIZATION

A thesis submitted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Javier Delgado

2007

To: Dean Vish Prasad
College of Engineering and Computing

This thesis, written by Javier Delgado, and entitled A Grid Computing Network Platform for Enhanced Data Management and Visualization, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this thesis and recommend that it be approved.

S. Masoud Sadjadi

Armando Barreto

Malek Adjouadi, Major Professor

Date of Defense: March 28, 2007

The thesis of Javier Delgado is approved.

Dean Vish Prasad
College of Engineering and Computing

Dean George Walker
University Graduate School

Florida International University, 2007

DEDICATION

This thesis is dedicated to my family.

ACKNOWLEDGMENTS

I would like to thank Everyone in my family for supporting me, inspiring me, and showing me the benefits of hard work and dedication. In addition, I thank my advisor, my committee members, and all of the good professors that I have encountered along the way, throughout the course of my studies. I would also like to thank all of the developers of the software that my work depends (and was developed) on; this includes The Globus Consortium, GNU/Linux, and many more. Last but not least, I appreciate the support provided by the National Science Foundation under grants EIA-9906600, HRD-0317692, CNS-0426125, and CNS-0540592 throughout my graduate studies.

ABSTRACT OF THE THESIS

A GRID COMPUTING NETWORK PLATFORM FOR ENHANCED DATA MANAGEMENT AND VISUALIZATION

by

Javier Delgado

Florida International University, 2007

Miami, Florida

Professor Malek Adjouadi, Major Professor

This thesis presents a novel approach towards providing a collaboration environment by using Grid Computing. The implementation includes the deployment of a cluster attached to a mural display for high performance computing and visualization and a Grid-infrastructure for sharing storage space across a wide area network and easing the remote use of the computing resources. A medical data processing application is implemented on the platform. The outcome is enhanced use of remote storage facilities and quick return time for computationally-intensive problems.

The central issue of this thesis work is thus one that focuses on the development of a secure distributed system for data management and visualization to respond to the need for more efficient interaction and collaboration between technical researchers and medical professionals. The proposed networked solution is envisioned such as to provide synergy for more collaboration on theoretical and experimental issues involving analysis, visualization, and data sharing across sites.

TABLE OF CONTENTS

LIST OF TABLES

TABLE PAGE

LIST OF ILLUSTRATIONS

ILLUSTRATION PAGE

1 Introduction

Medical professionals and technical researchers in engineering, computational, and other sciences have often collaborated throughout their careers. The synergy between them could engender a powerful force in solving medical problems. Advances in communication technology, such as electronic mail, file transfer protocols, cyber-infrastructure have proven to improve the efficiency of their collaboration efforts significantly. However, the newer technologies in distributed computing and networking ability which have become available to enhance this type of collaboration have not been fully exploited in seeking better solutions for efficient data sharing algorithms, visualization mechanisms, and computing capabilities.

The goal of this thesis work is to provide a platform for collaboration that takes advantage of the latest technologies available. In doing so, the concept of Grid computing, along with several available software packages for visualization, will be combined and configured to realize the desired platform. One of the principle uses of Grid computing is collaboration. In [1], Ian Foster (the "father" of Grid Computing), states,

"The sharing that we are concerned with is not primarily file exchange but rather direct access to computers, software, data, and other resources, as is required by a range of collaborative problem-solving and resource-brokering strategies emerging in industry, science, and engineering."

Also, in [2], they state that a critical requirement in a distributed, multi-organizational Grid environment is for mechanisms that enable interoperability. When combined with

visualization software, collaboration is enhanced even more. Such a platform implementation will be the focus of this thesis.

The concept of Grid computing has been very well received in the field of computer science. Several big companies, such as Sun Microsystems and IBM, have invested into its future. Furthermore, other scientists which commonly collaborate with computer scientists have become interested in the technology as it offers broader collaboration. The reason for this is driven by the fact that the amount of data that is generated every day has overwhelmed the scientific community. As a consequence, there is a desperate need for high-performance computing power and storage space, since many areas of science, such as medicine, particle physics, satellite technology, and radio telescopes, to name a few, are also becoming data-intensive.

The work outlined in this thesis will make use of the Globus Toolkit, version 4 (a.k.a. GT4 or just "Globus") which is the most popular and complete middleware for building Grid services. Globus provides many of the utilities an organization would need to start/join a Grid, including mechanisms that ease the sharing of processing power and storage space. The user is left with the task of building higher-level software specific to their clients' or organization's needs on top of this middleware.

GT4 is the premiere implementation of the Open Grid Services Architecture (OGSA). The OGSA is a proposed architecture, by the same people who developed Globus, which gives implementation recommendations and standards for Grid-enabled software. They promote the use of Web Services for implementation of services provided by the Grid, stating that a service-oriented view allows developers to address the need for standard

visualization software, collaboration is enhanced even more. Such a platform implementation will be the focus of this thesis.

The concept of Grid computing has been very well received in the field of computer science. Several big companies, such as Sun Microsystems and IBM, have invested into its future. Furthermore, other scientists which commonly collaborate with computer scientists have become interested in the technology as it offers broader collaboration. The reason for this is driven by the fact that the amount of data that is generated every day has overwhelmed the scientific community. As a consequence, there is a desperate need for high-performance computing power and storage space, since many areas of science, such as medicine, particle physics, satellite technology, and radio telescopes, to name a few, are also becoming data-intensive.

The work outlined in this thesis will make use of the Globus Toolkit, version 4 (a.k.a. GT4 or just "Globus") which is the most popular and complete middleware for building Grid services. Globus provides many of the utilities an organization would need to start/join a Grid, including mechanisms that ease the sharing of processing power and storage space. The user is left with the task of building higher-level software specific to their clients' or organization's needs on top of this middleware.

GT4 is the premiere implementation of the Open Grid Services Architecture (OGSA). The OGSA is a proposed architecture, by the same people who developed Globus, which gives implementation recommendations and standards for Grid-enabled software. They promote the use of Web Services for implementation of services provided by the Grid, stating that a service-oriented view allows developers to address the need for standard

interface definition mechanisms, local/remote transparency, adaptation to local OS services, and uniform service semantics [2]. Additionally, they say that service oriented architectures (SOA) provide "virtualization," which is the process of encapsulating the fact that operations are being executed remotely. This is true when compared to object oriented programming since SOA-built objects can interact via services running on several distributed computers.

In this thesis work, data virtualization will be provided by making transparent the fact that the medical data is distributed across several sites. As part of this thesis, several services will be built on top of Globus in order to provide useful functionality for achieving the specified goals.

As an example of how a Grid can be beneficial, consider the fact that typical workstations these days come with at least 40 gigabytes of storage space. Of this space, only about 25 percent is used for installation of the operating system. Additional applications for database, word processing, spreadsheets, image processing, etc. usually take an additional 10 - 25 percent. Data files, in the case that they are left on the workstation itself as opposed to a more secure, centralized file server, typically take very little space. As a result, at least half of the space of these drives is wasted. For example, a popular set of software that installed on systems is Microsoft Windows XP and Microsoft Office. These alone still only require 7-8 gigabytes of space, which is even less than 25 percent of the 40 gigabytes. Even if it is rounded to 10 gigabytes and another 10 gigabytes are used to store data (which is a lot for a typical workstation with office use only), it still leaves 20 gigabytes unused. This is quite a predicament since on one hand you have millions of com-

puters with several gigabytes of unused space, and on the other you have scientists eager to have more secondary storage space to store important information.

Another problem that exists is the lack of computing power for certain scientific applications such as in medical imaging, signal processing, and bioinformatics. There are certainly plenty of high-performance machines available, but many times using them is not easy, and Grid computing addresses this problem as well.

Another thing to consider, specifically for medical environments, is that nowadays many medical instruments require a lot of power for image/signal processing, as well as a lot of data storage requirements to store all of the acquired data. However, since these high-performance devices are not always in use, and because they may take up precious space unnecessarily, this creates a good circumstance for using Grid computing.

Aside from the standard benefits of Grid computing, a medical data storage Grid will provide a step forward towards enabling electronic storage of medical records. Even local newspapers are touting the benefits of such a need [3]. They cite that health-care advocates are encouraging the switch to computerized medical records. In fact, they even mention that the current president has stated that he wanted all Americans to have an electronic patient record by 2014. Supporters say that this will speed up medical decisions, avoid errors, and even save lives!

Thesis Statement: *Grid computing provides a way of enhancing collaboration amongst organizations. It makes it possible to solve computationally-intensive tasks on high-performance machines located in distant locations.*

To demonstrate the use of Grid computing for collaboration, the following tasks were accomplished:

- Deployment and configuration of a 16-node compute cluster, with a 15-tile mural display wall connected to it, demonstrating Grid-enabled high-performance computing and high-resolution visualization.
- Implementation of a practical application of Grid computing in a medical environment, with grid services created to demonstrate its power in allowing users to share storage space. This will be further extended to show how a Grid can be used to mimic a kind of "next generation operating room," in which each room could have a simple computer (or just an acquisition device) that is Grid-enabled, and all the data that it collects and needs to process can be sent to other members of the Grid for doing so. The software implementation will consist of a set of interacting GT4 services for distributed data storage.
- Practical experimentation of how remote users can easily use the power of remote supercomputers, by tackling a computationally-intensive application in brain research. Specifically, a Message Passing Interface (MPI) program is written that uses the Correlation Integral to detect when epileptic seizures occur. The Globus software that is installed provides tools for sending these applications from a local machine to be computed remotely.

2 Related Work

Several organizations have implemented similar collaboration environments amongst life scientists of different specializations, medical professionals, and computer scientists and engineers. One such project is *AliEn* [5]. *AliEn* attempts to provide a lightweight, but highly-functional Grid. It provides the following functionality:

- Support for a large number of files
- *MySQL* data catalog
- "Pull model" job submission – Resource broker which attempts to send jobs where data are; instead of locating resources, uses resources advertised by clients (i.e. computing elements.)
- Lightweight Directory Access Protocol (LDAP)-based remote configuration support for meta-data management
- Web-based monitoring and management
- Secure file transfer and replication

AliEn uses a service-based implementation, using a set of *Perl* Modules, which are wrappers around lower-level software, such as GT4 security commands, Andrew File System passwords, and Secure Shell (SSH) keys. It provides authorization and authentication by means of an LDAP-based database which controls and configures the following Virtual-Organization specific roles: people, roles, packages, grid partitions, and sites. *AliEn* uses a database structure that mimics the hierarchal nature of a file system: each directory gets its own table, and each table has its own structure.

As can be seen, *AliEn* provides the two main functionalities of Grid computing: distributed compute power utilization and distributed data storage. It does so using a lot of third-party software, including GT4, several open source *Perl* modules, and OpenLDAP.

The work presented in this thesis has several similarities to *AliEn*. For one, both extend on currently available software, although the work implemented in this thesis is based primarily on the Globus Toolkit. This is advantageous since most (and eventually all) of the software used by Globus is based on accepted Grid computing standards, i.e. the Open Grid Service Architecture (OGSA). This makes the chances of software developed a lot more likely to work well with other software built on the same paradigm. For example, *AliEn* uses the *bbftp* protocol [7] for file transfer. This is a suitable protocol for *AliEn*'s intentions. It is actually optimized for file transfers involving files over 2 Gigabytes in size, which is a good thing for many scientific applications. It provides on-the-fly compression, security, and authentication. The only notable drawback is that it is restricted to Unix-compatible environments (thus making *AliEn* restricted to this environment). The foreseen problem is that since *bbftp* is not a standardized or widely available protocol, other workstations are less likely to have it installed. Globus ships with the *gridftp* protocol. As a result, any system with Globus can send and receive *gridftp* transfers. One could argue that Globus, too, must be installed, but it is easier to just have to worry about installing one software package than several. Furthermore, the possibility of there being conflicts amongst the software components is virtually eliminated if they are all being distributed by the same organization (who would presumably test all of its components working together) and certified by the same standards board.

Following on the example about *bbftp* versus *gridftp*, *gridftp* was originally only available for Unix. However, since there has been so much development in Grid computing and the need for *gridftp* on the Windows platform, researchers at the University of Virginia have implemented a version for Microsoft's .NET platform [8].

Now that *AliEn* has been discussed, it would be useful to describe a project, similar to the one to be implemented as part of this thesis, which uses *AliEn*. *AliEn* is being used in several projects in physical sciences and various other fields, but the one most comparable to this project is the *MammoGrid* project [9]. The *MammoGrid* is a European-Union-funded project for distributed mammogram analysis. It consists of several mammogram-specific grid services, which integrate with the Web Service Resource Framework (WSRF), which is the second (as of this writing) implementation of OGSA. OGSA is a specification set fourth by Globus and others to come up with a standardized way of building Grid computing services, using a service-oriented architecture (SOA). The purpose of the *MammoGrid* software is to provide data capture (of records), management, and storage of huge files for fast retrieval, comparison, and diagnostic review. Like the software implemented in this thesis, it works with Digital Imaging and Communications in Medicine (DICOM) files and uses Java for all of its business logic. The main differences are that it uses *AliEn* for its Grid-related functions, such as data transfer, and it deals specifically with mammogram files.

Another storage-only implementation that is currently available is the Storage Resource Broker (SRB) [10]. The SRB provides access to different types of data storage across the grid and provides meta-data (using its meta-data catalog, or MCAT feature) for all stored data. This meta-data is used for describing the data in order to be able to query and dis-

cover it. The meta-data is also used to describe the location and ownership of files. It accepts various types of data, including Unix directories, Windows directories, binary large objects (BLOB) stored in databases, and regular database objects. One feature that SRB has that the others do not mention anything about, which will also be implemented in this thesis work, is the accessibility of files based on meta-data rather than actual names and locations. Another unique feature of the SRB is the fact that it is designed to be used with various types of databases. As will be described in the Implementation section, using a third party, open source, grid-enabled database implementation, this feature will exist in the software developed as part of the thesis.

A fundamental concept in any Data Grid is privacy. In this case it is especially important since medical records contain personal information about patients that should only be readable by authorized users. Even local system administrators should have reduced access to these records. One approach taken towards this is the Distributed Medical Data Manager (DM2) [11], which proposes an architecture where image files are encrypted before being sent to the Grid. They do not mention how meta-data queries can be made, while taking the privacy issue into account. Without such a mechanism, anyone can make a query for any patient's records.

Since the SRB, like *AliEn*, only provides data-grid functionality, some of the various projects using it, including a couple in neuroscience, should be discussed. The Biomedical Informatics Research Network (BIRN) is possibly the largest project using the SRB [12]. It is a platform for collaboration amongst several educational and medical institutions. The idea behind it is to provide biomedical researchers with large amounts of data obtained from the various medical facilities. In their original paper [10], they describe

briefly how they went about solving the authorization/privacy issue for BIRN. They satisfied BIRN's requirement of applying access controls on both the images and the descriptive and administrative meta-data registered onto the logical name space. The logical name itself had the access controls applied to it. In order to moderate access to meta-data, the SRB was modified to extend on the traditional "read, write, execute" and "users and groups" permission scheme allowed by modern operating systems.

The goal of the Data Grid portion of this thesis work is to provide much of the functionality of the aforementioned projects, but using only Globus-supplied components. It is not possible within the context of this thesis, given the limited time frame, to go into the level of detail that these other projects did. For example, *AliEn*, when used with the *AliEn*FS module [13] allows their files to be accessed, stored, and manipulated using regular file system commands. SRB does this as well, they specifically mention supporting the following file operations: *create, open, close, unlink, read, write, seek, sync, stat, fstat, mkdir, rmdir, chmod, opendir, closedir*, and *readdir* operations [10]. *AliEn*FS doesn't specify the commands that it implements, but it uses the Linux File System in User Space (FUSE) kernel module, which requires the following functions to be implemented: *getattr, readlink, getdir, mknod, mkdir, unlink, rmdir, symlink, rename,link, chmod, chown, truncate, utime, open, read, write, statfs, flush, release, fsync, setxattr, getxattr, listxattr, removexattr* [14]. One should keep in mind that the *MammoGrid* was a three-year project, and it used *AliEn* (unlike the work done here, which implements all of its own functionality); *AliEn* took one year to have its initial prototype (and *AliEn*FS came later). Also, both projects had several developers. Furthermore, this implementation will provide some improvements over their designs. In fact, in [15] Ian Foster states that the SRB

does not comply with one of his three requirements for a certain piece of software to be considered suitable for Grid computing. That criteria being that the software should use standard, open, general-purpose protocols and interfaces. The project associated with this thesis will implement its own functionality using only Globus Toolkit-supplied services and a couple of other open source software libraries.

As far as visualization is concerned, there have been several organizations who have built display walls for various purposes, including collaboration and high-resolution visualization. The typical solution is to use the Distributed Multihead X server (DMX) software to form a single large display from the shared displays of several nodes in the cluster, which is controlled by the head node. This is known as the "metabuffer" concept [16]. Among them, the Scalable Adaptive Graphics Environment (SAGE) [17], whose software may be utilized as part of this work, built a collaboration environment called "LambdaVision" They don't cite any information about sharing data nor about any specific kind of visualization being used. But they do provide the software that they used to implement their wall. SAGE provides an alternative to the typical mural display software. They use their own software to communicate amongst the display nodes and display images. One consequence of this is that only SAGE-enabled applications can be run on a SAGE display. However, many benefits are provided, such as the ability to share regular and tiled displays of different sizes and the ability to control the tiled display from any other system by using a client interface.

The other topic that will be addressed, which deals with epileptic seizure prediction and detection using electroencephalogram (EEG) data, has been researched for quite a long time. Various procedures have been researched for performing seizure detection and pre-

diction, including frequency-based methods, statistical analysis, and non-linear dynamics, such as wavelets, correlation integrals, cross correlation, phase, and principal components [4] [5]. One of the reasons for the lack of substantive studies has been the lack of processing power and storage space needed for analyzing EEG data, which span gigabytes in length [5] [6]. To address this, a parallel-processing routine that can run on a Grid-enabled compute cluster is implemented.

3 System Design

3.1 Infrastructure

A 16-node compute cluster and several workstations are used as part of this Grid that has been built here at the FIU's Center for Advanced technology and Education (CATE). In addition, the systems that form the Latin American Grid (LA-Grid) [20] will also be participants. The cluster and mural display are shown in illustrations 1 and 2, respectively.

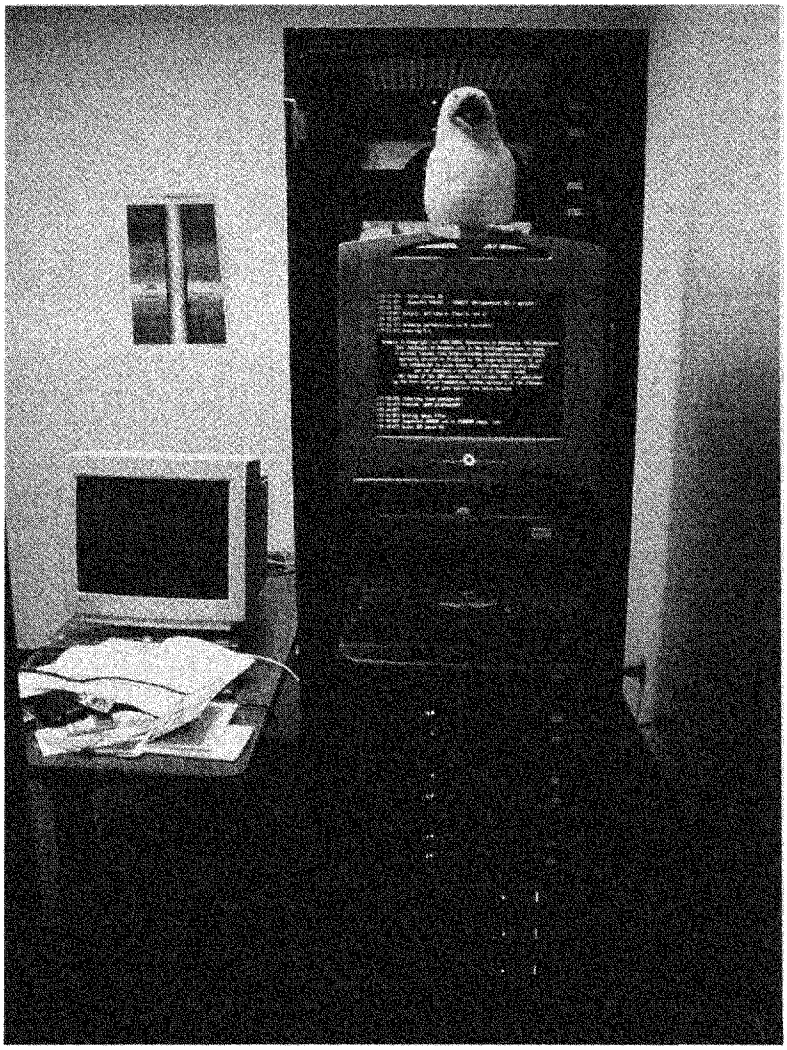

Illustration 1: 16-node compute cluster and mural display controller

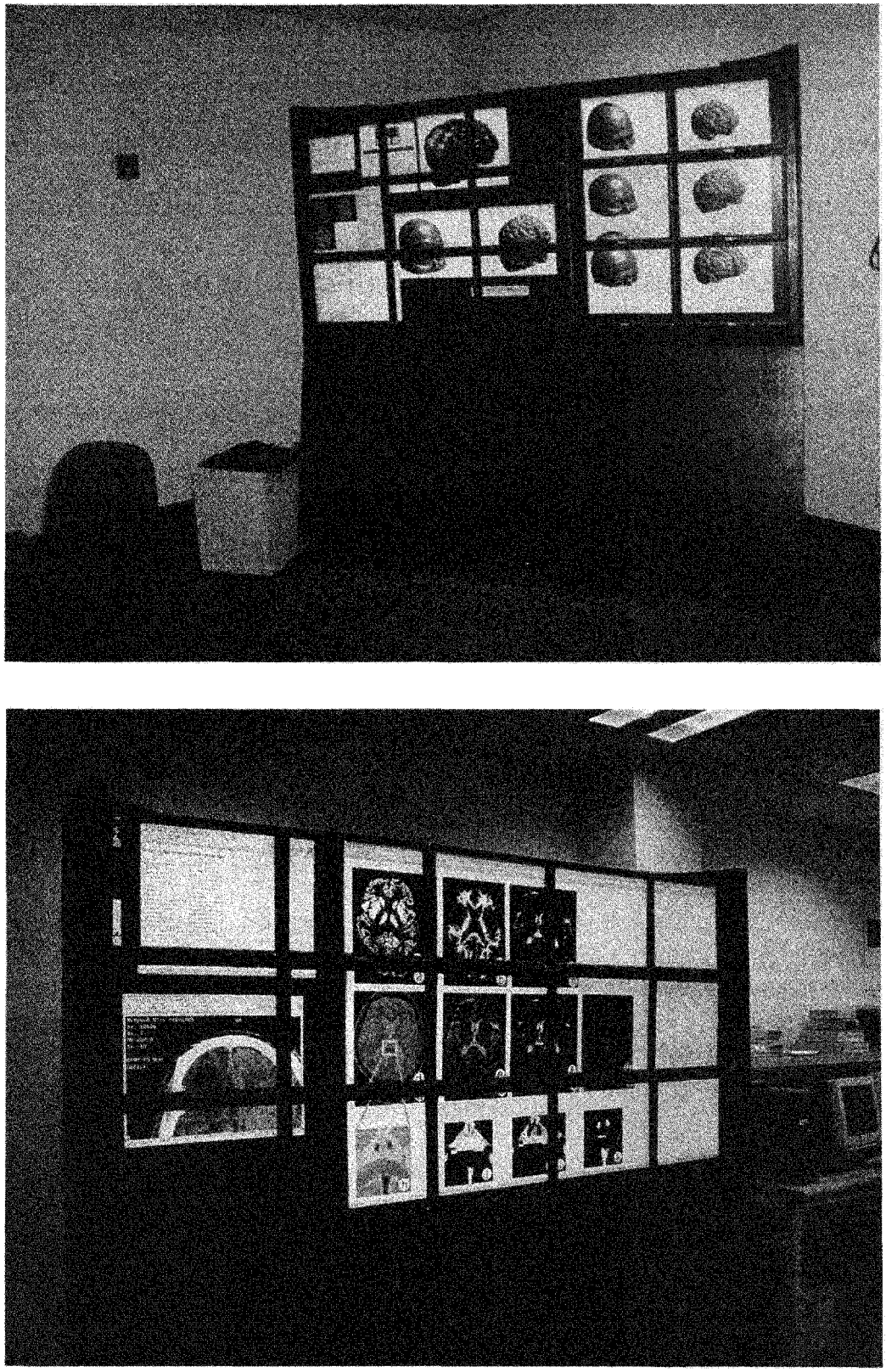

Illustration 2: 15-tile mural display

The 16-node cluster hereafter referred to as "Mind" will be the main testbed for implementing this thesis work. In addition to being used for the Grid software, it also serves as the backbone of a 15-tile mural display. Mind's resources are available to local resources. When not serviced on-site, these resources are also accessible to Grid-members.

While modern clusters are not supposed to be especially difficult to deploy thanks to modern Linux Operating System distributions that facilitate their management, several issues must be addressed to get the complete system set up. Also, since the mural display is also integrated into this system, some added complexity is introduced.

The Operating System (OS) distribution that came with the cluster is Platform Rocks. Rocks does a good job at automating many things; it provides functionality for easily adding nodes to the cluster after the OS distribution has been installed on the head node. It also provides tools for running commands across the whole cluster, such as adding and removing software packages. There was, however, one problem when upgrading the operating system that was preventing the head node from installing compute nodes. This turned out to be a bug in the operating system distribution that had a hard-coded timeout set when broadcasting an IP address to compute nodes to install them; as a result, compute nodes were not given enough time to boot up using the network boot image from the head node. This was fixed by setting some network parameters to the management switch used to connect all of the nodes in the cluster.

When everything was set up, all necessary packages were verified. A few of them needed some configuration. The Message Passing Interface (MPI) software, for example, which is paramount for clusters, needed some initial set up to work correctly. Rocks came with

two major MPI version 2 implementations: MPICH and LAM. Neither one worked initially because none of the necessary files for compiling or running MPI programs were installed in the system path. Therefore, it was necessary to change the default login profile that is read before each user is logged in. Also, it was necessary to create a file that tells the MPI programs which machines are available for running MPI tasks on.

Surprisingly, Rocks does not do this automatically. In addition to MPI, a scheduler is needed in order to queue jobs for cases in which there are users competing for compute resources. There are several schedulers available, although they work much the same. LSF was chosen as the scheduler for the cluster, since Platform provides a user-friendly GUI for submitting and viewing jobs using LSF. With MPI and LSF set up, the cluster was ready for high performance computing jobs.

Clusters are popular targets amongst "black hat" [1] hackers since these users need a lot of compute power for a lot of the "work" they do, such as cracking encrypted passwords. In fact, Mind was compromised within the first week of ownership. This required a complete clean up and re-installation of the whole system, to ensure that no traces of malicious software nor open gateways were left on the system. As a result, some measures had to be taken in order to keep the system as secure as practically possible.

No intricate security scheme was induced, as there was no time for that and it would burden the usability of the system. However, a few measures were taken to prevent future attacks. The most important change was disallowing remote logins for the administrative (root, in Unix-speak) user. This way, brute force attempts to log in remotely as root

1. "crackers," also known more generally as hackers are usually separated into "black hat" and "white hat" types. The former being the ones with evil intentions

would not be possible. Therefore, the only way to gain root access from a remote site would be for the user to figure out the user name of one of the regular users, obtain his/her password and log in. Having to guess a user name *and* a password is extremely difficult. Therefore, that only leaves the attacker the option of having to go in through some security hole in the system to perform a remote login. In order to prevent this from occurring, the system was signed up with the Red Hat Network, which is a service available to users of Red Hat's enterprise software, which sends email updates in case security vulnerability is discovered in any software package. Patches or software updates to resolve these problems are then promptly done.

In addition to that, the system now emphasizes "strong" passwords (i.e. passwords that are not based on simple dictionary words) for all users. Also, there is no FTP daemon running, as FTP is an insecure protocol for transferring files. Instead, users can transfer files using the SCP/SFTP protocol, which works over the secure protocol, SSH. This is a bit of a burden to some users, since they are probably accustomed to using tools designed for FTP, but higher security will always result in at least a little added inconvenience for the user. In this case, it was a worthy trade-off.

3.2 Visualization Platform

3.2.1 Motivation and Necessary Software

Given the nature of collaboration between technical researchers and medical professionals, remote visualization would greatly enhance such an environment. Being able to share work, i.e. desktop display(s), remotely while at the same time teleconferencing or video

conferencing will take collaboration efficiency to a whole new level. It may even reduce transportation costs since many more interactions will be possible remotely.

The goal for the visualization portion of the platform is to allow stadium-style teleconferencing while simultaneously enabling the sharing of desktop displays. This way, using a tiled display wall, several collaborators can view and share their work at the same time, no matter where they are. For example, a doctor may be displaying a high-resolution magnetic resonance imaging (MRI) image of a brain and demonstrating to his/her collaborators an abnormality in a specific section of the image. The technical staff can perhaps then perform some image processing routines on that same image, and share any findings with the doctor.

Implementation of the aforementioned visualization platform requires several steps. Creating a compute cluster has been made easier thanks to tools such as the Rocks distribution. However, tiled displays still require quite a bit of work before they are fully operational and efficient functionally. First, a mural display will have to be physically built and then configured using the appropriate software. The display configuration process will take several steps, since several software packages are needed for different functions, decisions must also be made about which is the best software package for the job.

The fundamental package required for the mural display is DMX. While some of the other software applications used could run without DMX, DMX makes their operation more transparent. This is a useful first step towards having a working mural display, but some additional packages are needed for more advanced visualization uses.

With a gigabit Ethernet connection, DMX renders 2-D images very quickly and system interaction is responsive [21]. However, for three-dimensional visualization, DMX performs very poorly, if at all. The problem is that DMX works by sending all display information through the network. With a gigabit Ethernet connection, this is fine for regular two-dimensional image rendering. Windows move fluidly and a 3100x1100 image requiring nine tiles (at 800x600 each) is rendered in just over a second [21]. However, for more graphically-intensive applications, such as three-dimensional visualization and video playback, it does not suffice, at least not with present network technology. To address the problem of 3-D rendering, a group of developers from Stanford created an application called *WireGL*, which was later spawned off to become *Chromium*. What *Chromium* does differently is, rather than sending display information, it sends OpenGL instructions to each display node for local rendering, which provides much better performance [22].

The final component that seemed necessary was software for providing improved remote collaboration. There have been a few developments in this area. For basic sharing of desktops, there are Virtual Network Computing (VNC) and *FreeNX*. There are also a couple of full-fledged collaboration environments: *AccessGrid* and SAGE [23]. When tested, VNC was found to work well for general desktop sharing, but has two drawbacks. For one, video reproduction is poor. This hinders collaboration in cases where video conferencing is involved. The problem with VNC is, like DMX, it sends all of the pixel information over a wide-area network, which is nowhere near the necessary bandwidth for direct rendering of video.

For example, to send a movie that is 1024x768, which is about 786432 pixels, with 16 bits per pixel, at a rate of 30 frames per second, would require a bandwidth of 377,487,360 bits per second, or about 47 Megabytes per second, much less than is currently available. For comparison, a T3 connection has a bandwidth of about 44.736 Mbits/s, or about 5.5 Megabytes per second. The other problem applies to the use case scenario in which two collaborators, both utilizing display walls, want to share their entire displays. The problem here is that VNC servers do not seem to work on entire walls. When attempting to do this, the server rejects the attempt, stating that a VNC server is already running on the display, even if it isn't. Apparently a conflict exists between actual VNC sessions and the video networking that is caused by DMX.

The newer alternative to VNC is *FreeNX*. *FreeNX* is designed to take advantage of X Windows' inherent network-ready design in order to create a more efficient way of providing a remote desktop access. Since X Windows is designed using the client-server software architecture, it is well-suited for the task of having the server portion (the part that allows windows to be drawn, moved, resized, etc.) on a separate, networked system. Unfortunately, *FreeNX* servers do not exist for Windows as of this writing. Since allowing connections from heterogeneous nodes is desirable, *FreeNX* is ruled out as an option.

Of the two full-fledged visualization and collaboration software environments, *AccessGrid* has the best feature set for remote conferencing and collaboration. It provides a developer API for creating "Shared Applications," which are applications built for collaboration. They use network protocols for submitting changes in the applications to all users in a "virtual venue" transparently [24]. For example, using "Shared PowerPoint," a lecturer could be showing a PowerPoint presentation and when the lecturer changes a slide,

the change is relayed to all collaborators. A virtual venue, in this scope, is a virtualized collaboration environment consisting of clients (regular users and administrators) and applications (such as presentations, video streams, etc.) [25]. This is done in addition to the built-in collaboration (i.e. video conferencing) software. Furthermore, *AccessGrid* provides security by means of Globus certificates to ensure that only authorized members can view the presentation.

The other collaboration software that was tested, SAGE, is more of a general-purpose solution for mural display deployment, with applications for desktop sharing, image and movie viewing, and more. SAGE makes these shared-application-like features possible as well, since any networked computer with SAGE installed can connect to and control a SAGE display. SAGE has a more limited set of applications, though. Also, with SAGE, the installation is a bit complicated and there is a learning curve associated with using it. When installing, it is necessary to also include an application called QUANTA (developed by the same group as SAGE). Neither application offers distribution-specific packages, leaving the task to users. Also, several configuration files need to be correctly set up.

An even bigger drawback is for ordinary users needing to use a SAGE environment, given the lack of familiarity of the interface. In addition, SAGE displays cannot be controlled directly, unlike DMX displays. Instead, they are controlled by other systems running an application, SAGEUI, which displays a smaller version of the mural display on a user's computer. The user can open new applications, move windows, chat with other users, and browse system files. The application is clever and even convenient, since controlling a traditional DMX display is difficult due to the large size of the screen and the

components of the window manager, but neither the look and feel of the display nor the applications are very intuitive, as can be seen from the screenshot in Illustration 3.

User-friendliness is of utmost importance when implementing a platform to be used by users of differing lines of work. As a result, it seemed like using a DMX display was a better option.

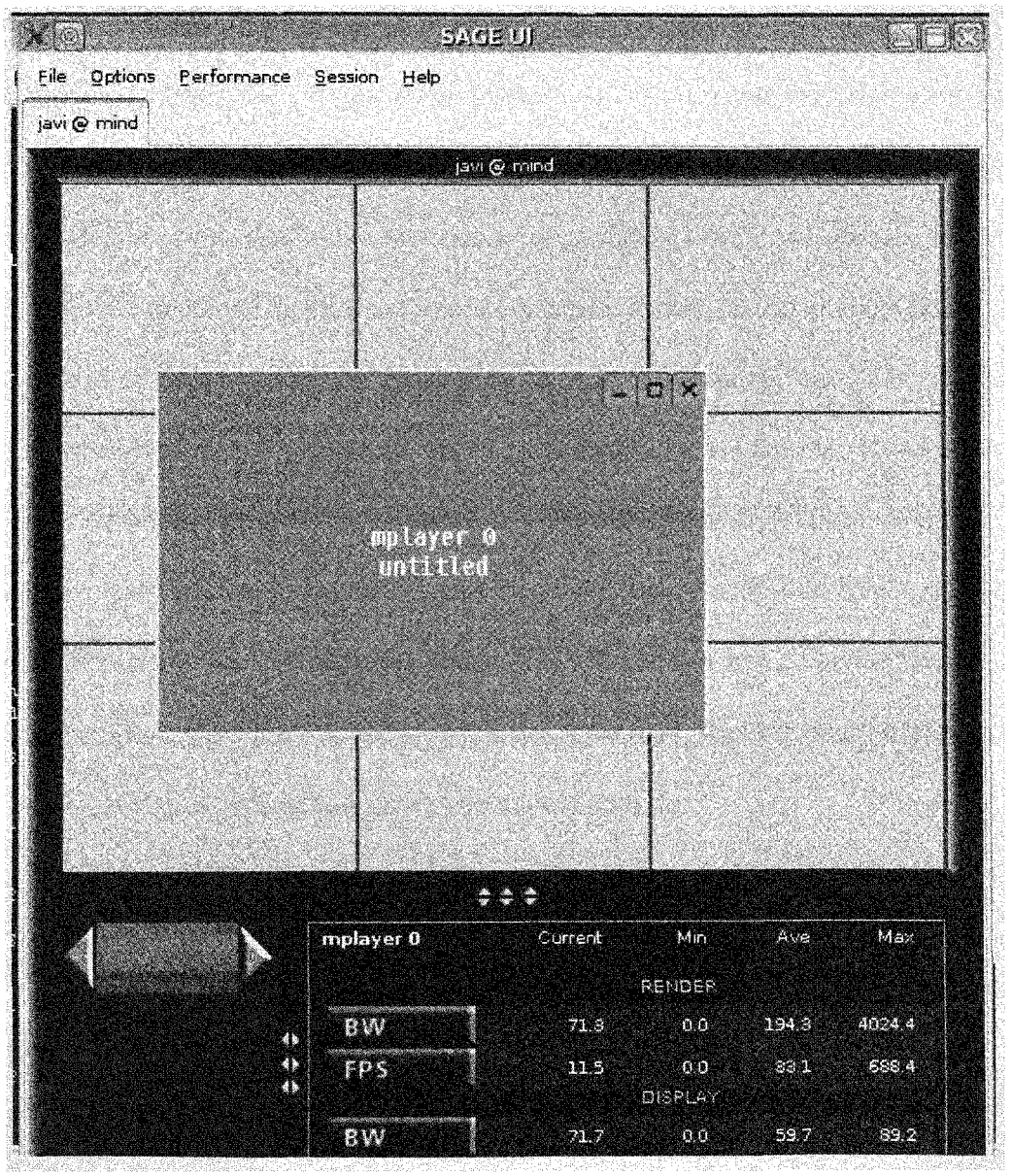

Illustration 3: Screenshot of SAGE UI

3.2.2 Implementation

The two major packages involved in setting up the mural display are DMX, for combining the displays from each node in the cluster as one giant screen, and *Chromium* for rendering 3-D graphics on the tiled display. Since mural displays are not very common (as compared to ordinary compute clusters), and thus there is not enough valuable documentation available for setting them up, a lot of trial and error and debugging was involved in this part of the project. The mural display was first set up under the original operating system distribution (Rocks 3.0); the process of starting up the display was very long and cumbersome. This was due to the fact that a lot of things were done quickly and incorrectly, just as a means of getting the system up and running (at that point it didn't seem reasonable to do things "correctly" since there were so many things that could go wrong). Everything worked, but it was not a very elegant setup. For starters, all of the necessary packages needed to be manually installed on each node (times 15 nodes). Furthermore, every time a user wanted to start the mural display, he/she would need to run commands on each node to log into it, start graphical mode (X windows), and allow the head node to access its display. This also entailed physically connecting the keyboard and mouse to said system, which is difficult on a rack-mounted computer.

As a result, getting the mural display on would take at least fifteen minutes, which is unacceptable. In addition, if a server goes bad or needs to be reinstalled for whatever reason, the whole mural display set up process would have to be done all over for the node. Therefore, something more automatic needed to be conceived. Having learned from the initial set up of the mural display, some research about the details of how Rocks clusters

work (and specifically, how compute nodes are configured) had to be done. Using this knowledge, a way of automating the process of creating a mural display using Rocks had to be devised. There is actually a ready-to-go visualization system available from the Rocks developers, called the "Viz Roll" [26]. However, it is only available for i386 platforms. Since future uses of the cluster may require the added pipelining of its 64-bit architecture, and since it was desired to get maximum performance out of the cluster, using the Viz roll was not a viable option. Another thing worth mentioning is that the Viz roll must be installed during an initial installation of the cluster [26], which would have required additional configuration time since the cluster was already configured.

To explain what needed to be done to get the mural display working, some information about how Rocks approaches cluster computing will be necessary. Rocks essentially stores an operating system distribution image, which reflects the image that gets installed on each newly-installed compute node, on the head node of the cluster. This "image" is actually composed of several configuration files, along with binary software packages, which are all sent to a compute node when it is installed (or "kickstarted" in RedHat-linux terminology). Kickstarting is a RedHat concept, which Rocks extends by using XML schema to provide a myriad of additional configuration possibilities). Therefore, the necessary step for creating a visualization cluster based on Rocks is to come up with a Kickstart scheme that will make certain configuration changes and instruct compute nodes to install additional packages in order to make them mural-display ready (i.e. make them *display* nodes). But first, it was necessary to figure out exactly what is needed to put into this kickstart XML file and also figure out exactly what packages need to be installed.

The following goals describe the desired outcome:

- All necessary packages must be installed when the operating system is deployed onto a new node.
- Users should be able to start the mural display from the head node, without having to interact directly with any compute nodes
- In case of server failure or OS re-installation, nothing besides the regular installation procedure should have to be performed.

With these requirements in place, one can start dealing with the system configuration files of a test node to get it to a working state. It is then necessary to figure out what software packages need to be installed onto this node. Realizing the final goal is essentially trial and error. The first thing that was done on the test node was to install several graphical client software packages. A desktop environment was needed so the Gnome desktop environment was chosen for two reasons: (1) it is the default environment for Red Hat Linux (and therefore should be the most compatible), and (2) it provides an automatic login feature, whose use will be described later. Gnome consists of several components. All of the ones that seemed necessary in order to install the display manager will be installed one by one on the test node; whenever an error about missing libraries or packages is encountered, the dependency (i.e. library or software package) will be installed and the process repeated until the display manager is finally installed and running as needed.

In order to start the mural display from the head node-without interacting with the compute nodes (goal number 2), the following conditions will have to be met:

- When a compute node is installed, after it boots up it must enter a graphical environment (as opposed to a command line interface)
- Once the graphical environment starts up, it must log the user in automatically
- The logged in user must allow the head node to access its display

Most of these steps were achieved by modifying system configuration files on the test node. This was a long process since there are literally hundreds of configuration files in the system. The key to doing this was overseeing which ones were being accessed by certain programs and scripts. To determine this information, a couple of system tools were used. The Unix "strace" command, which helps by printing a trace of system calls used by an application, was used for binary programs. The system calls helped by showing what libraries, executables, or files were being looked for (and where). Scripts were invoked with verbose output from the shell. For example, when the "startx" command (which is a script that starts X windows, the Unix graphical environment) is run, it looks at the user client file, the user server file, the system client and server files, and others. Furthermore, it searches for these files in system-wide directories and also in user directories. In the end, it turns out that these were the files that needed to be modified:

- */etc/inittab* – this is not specifically a graphical configuration file; it is the file that is read by the system just before starting its services. It tells the system what runlevel 1 to start at. It was changed to boot to runlevel 5, which starts all of the graphical-mode applications. To accomplish this, the Linux "awk" utility was used to read through the file and change the line that instructs it to start at runlevel 3

(which is command-line mode) and tell it to instead start at runlevel 5. This, of course, is carried out after the node has been successfully set up.

- */home/mural/.Xclients* – this file is executed when user "mural" starts X Windows. "mural" is simply a dummy user that was created to host the graphical environment. This was done as a security measure. A separate account with very limited privileges leaves fewer possibilities for intentional and unintentional damages to the system. In the *Xclients* file, commands were given to ensure that the head node could access the compute node's display, to turn off the screen saver, and to use the most basic window manager possible, *mwm* in this case, in order to use as little resources as possible.

- */etc/X11/gdm/gdm.conf* – this is the configuration file for the Gnome Display Manager (GDM). This file had to be modified to automatically login the mural user. This needed to be done since the only way for a computer (i.e. the head node) to access the display of another computer (i.e. a compute node) is if a user is logged in and accepting requests from other hosts to access (or totally take over) its display.

In addition to all of these issues, the X-Windows configuration file (*xorg.conf*) from the head node had to be included in the compute nodes' image. The X font server had to be enabled for runlevel 5. All these changes were specified in the *extend-compute.xml* file, which is the last file that Rocks reads when deploying a new node. Then, all of the external files and packages referenced in the configuration file were placed in a specific directory.

Once the configuration was finished, a new distribution was built using the rocks-dist command. It was then tested by reinstalling the test node to see the outcome. Several attempts were made and errors were encountered before reaching the desired state. After that, the fifteen other nodes were reinstalled.

The next step was configuring files on the head node for the mural display. The DMX application uses (in addition to the regular X files) a configuration file that tells it what hosts (i.e. compute nodes) to use for the tiled display, and how to arrange them. A script was also created to start the mural display. This script takes care of goal number 3 above.

The next phase of the mural display setup was installing *Chromium*, which is used for 3-dimensional image rendering on tiled-display walls. *Chromium* can be made to work by itself or to interact seamlessly with DMX. The latter is harder to get working but is a more elegant outcome. The latest version of *Chromium*,1.9, was used. Unlike DMX, *Chromium* needs to be installed on all of the compute nodes, not just the head node. Therefore, it was necessary to build an RPM that can be included as part of the "kick-start" installation for compute nodes. Before doing this, it needs to be compiled successfully.

Getting *Chromium* to compile did not go very gracefully. The initial attempt gave the following error: "R_X86_64_32 against 'a local symbol'." Some online research lead to the conclusion that this was due to not having compiled DMX with a certain parameter, namely "-fPIC." Looking through the documentation for the GNU C Compiler (GCC) revealed that using this option results in position-independent code, suitable for dynamic linking." [27]. Furthermore, according to [28] PIC allows built objects to be placed any-

where in the process' address space, unlike traditionally-compiled objects. Apparently, since *Chromium* relies on certain DMX libraries when it is to work on DMX-enabled displays, it needs said libraries to be compiled with PIC enabled order to be able to link to them. The DMX distribution that was installed on the cluster was a binary version which apparently wasn't compiled with this parameter. As a result, DMX had to be recompiled.

Compiling a custom DMX package turned out to be a bit complex as well. The problem being that Red Hat provides a binary package with just DMX but they don't provide a source-version. Only the entire X server can be rebuilt all at once from source. The following course of action was taken to address the problem:

- Downloaded RedHat-provided source code for the X server and extracted it
- Passed "-fPIC" to the standard compiler options in *SOURCES/xc/config/cf/linux.cf*
- Recompiled using the "rpmbuild" command
- Removed old xorg-packages
- Installed the newly-created RPM packages.

Unfortunately, although the procedure itself was successful, the same compilation error occurred when attempting to compile *Chromium*. Having failed at that, the next option was to try installing only the DMX files from the source code provided directly by the DMX project and see if it works with the existing X packages. As it turns out, the source code provided by the DMX project includes a lot of standard (non-DMX) X code. However, using all those files didn't seem like a good idea; for the sake of security updates and more efficient system administration, it is best to use distribution-specific packages

whenever possible. In the case of Mind, this means using Redhat-provided RPMs. This way, software updates and security patches can all be found in one place (i.e. the Redhat network). In order to use the custom DMX along with the Redhat RPMs for the rest of the X packages, the following steps were carried out:

1. Downloaded and extracted the DMX source code
2. Modified the host.def file: passed the -fPIC parameter to the compiler
3. Compiled it
4. Noted what files the Redhat-provided RPM of XDMX (using the rpm command)
5. Copied those files, but the newly-compiled files matching (from 3) to those matching the ones in 4, using the same directory structure specified by 4.

Since this was a non-standard way of setting up the software, DMX had to be run again to ensure it was working. After doing this, it became time to compile *Chromium* again. This time, it compiled successfully. The compiled *Chromium* was then copied to the other display nodes. Finally, *Chromium* was tested by running an OpenGL using *Chromium* manually.

The last step is now to get *Chromium* to work seamlessly with DMX. This entails modifying a couple of *Chromium*'s configuration files, as per the user manual. The only problem encountered in this part, was in the *.crconfigs* file, which is responsible for calling *Chromium* when an OpenGL file is executed. The line that calls the *autodmx*.conf was tab-delimited, but apparently whatever *Chromium* scripts processes it only parses spaces. After switching from tabs to spaces, everything was working properly. This was confirmed by re-running several OpenGL applications, including the popular "glxgears" and "atlantis" programs.

Once all the visualization software was correctly working, the collaboration software had to be installed and configured. After trying out both SAGE and *AccessGrid*, it seemed like *AccessGrid* would provide better functionality. First of all, with SAGE, only the display control software can run on Windows operating systems, so it would not be possible to use SAGE applications (i.e. for video conferencing) on a Windows system. A Windows display could be shared using VNC, but as explained previously, this provides unacceptable video performance. Also, the *AccessGrid* suite provides more collaboration software, such as SharedPresentation.

An attempt was made to use the latest version (which is 3.0) of *AccessGrid*. However, it was unable to fully install using version 2.3 of Python, which is what is supported by Rocks 4. Upgrading python was not feasible since so many system packages depend on having a certain version installed. As a result, version 2.4 of *AccessGrid* was used.

3.3 Configuring the Globus Toolkit

Grid-enabling Mind will require quite a few steps. First, the Globus Toolkit must be installed. This involves compiling the source code, installing it, setting up optional services, and setting up security. After it is installed, it will be necessary to get familiarized with the framework and various programming interfaces for GT4-provided tools. It will also be necessary to find an available API for dealing with DICOM files. Based on the steps executed here, an overall design for the project will be conceived.

It will be worthwhile to give an overview of the installation and configuration process of the Globus Toolkit. Installing the toolkit itself is trivial as long as all of its dependencies are met. However, some additional steps are needed for some of its services. For exam-

ple, to use the Reliable File Transfer (RFT) service, which will be needed for fault-tolerant file transfer, it will be necessary to set up a database. The RFT uses a database in order to keep state information about the status of a transfer, in case an unexpected error occurs in the server [29]. This obviously requires a database server to be available on the system. The database server will be a requirement later on in the project also, for storing the DICOM meta-data. The cluster already had *MySQL* configured, so it was chosen as the database for RFT as well. Therefore, all that had to be done is create the database and configure RFT to use it.

Another component of GT4 that must be installed separately is the Replica Location Service (RLS). RLS will be necessary in order to map logical names to physical file names on the data grid. RLS will need to be compiled and installed separately, as it is not part of the default GT4 installation. RLS requires an Open Database Connectivity (ODBC) driver present in the system. By default, it uses the *iODBC* implementation. However, after installing this on the cluster it had problems working, apparently due to an incompatibility with the version of *MySQL* installed. Fortunately, GT4 provides functionality with the *unixODBC* implementation of ODBC as well. After rebuilding the RLS with the *unixODBC* option enabled, some RLS tests were run using command line clients to create, query, and remove logical name mappings to physical files. All the tests were successful. Next, since the RLS interaction takes place in the service code, rather than on the command line, it was necessary to create a test program for performing these same RLS tasks.

Writing the test program required searching through the available public interfaces for RLS. After implementing a program (using Java) to perform the same functions done in

the command line, one would think that everything would work the same. Surprisingly, this wasn't the case. The problem turned out to be due to a bug in the GT4 code that attempts to look for 32-bit libraries, even if it is running on a 64-bit system. Fortunately, there were patches available from Globus; once they were applied and the code recompiled, the API tests were successful.

The final part of the system-administration part of the project was getting the GT4 security layer (i.e. the Globus Security Infrastructure or GSI) configured. This entails setting up a certificate authority, which Grid users need to authenticate with every time they need to use the Grid resources. Also, Grid certificates need to be set up for the system account (i.e. the globus user) and for all users who will be using the Grid (i.e. the members of the virtual organization). For the certificate authority, the Globus-supplied *SimpleCA* software was used. The same process was done for all of the workstation nodes that are to be part of the virtual organization. In order to prevent having to copy the user certificates to all of the workstations, *MyProxy* was set up on Mind. A *MyProxy* server keeps certificates locally and allows remote users to access them [30]. To be able to work with the LA-Grid nodes, which authenticate with a separate certificate authority, the user certificates from Mind were transferred to LA-Grid, and vice versa. Also, the grid map file, which Globus uses to map Grid users to local user accounts (i.e. Unix accounts), on each system needs to have the accounts from the other system.

With all of the software set up, it was time to start working with the programming interfaces provided by Globus. Globus provides some documentation in the "Public Interface Guide" sections of their online documentation. This was necessary when creating classes for interacting with the RFT and RLS. Additional information for implementation of the

services are obtained from "Globus Toolkit 4: Programming Java Services" [31]. As mentioned in the introduction, OGSA recommends a service-oriented implementation approach for Grid services. Therefore, all of the functionality for the *DataGrid* and DICOM services are implemented as Web Services, i.e. WSRF web services.

The term "resource," when mentioned in the scope of Grid computing (and particularly WSRF), can be quite misleading. The term probably originated from the fact that the resources that were to be published in grid services are physical resources, such as CPU and storage information. However, "resources" in services can be anything that allows storage of state information for the services. For example, the Globus Reliable File Transfer service has as resources percentage of files transferred and number of files being transferred, among other things. These aren't exactly what one would think of the term "resource", thus this brief explanation of the ambiguity.

3.4 Initial Service Implementation

Two major services are involved in this project. One is the *DataGrid* service, which will provide the ability to share files on the grid. The other is the *DicomWriter* service. DICOM is the standard format for transmitting medical image data [32]. The latter service will portray how a service can be used by an acquisition device (such as an MRI Scanner) to capture data and send it to the Grid. When these two services are combined, they provide a general user, such as a medical expert, a way to utilize the power of Grid computing in a transparent manner. Illustration 4 portrays the basic infrastructure scenario involved in a medical situation: the nodes in the operating room connect to the data nodes (*SAN_1* and *DataKing*) to store data. As a result, the nodes in the operating room don't

need to have very large hard drives (if at all). Furthermore, if the data later needs to be processed, it will already be in a high-performance facility. The flowchart in Illustration 5 demonstrates the typical flow of events that is being implemented. The two rectangles with hatches represent the Grid services. In this case, they are actually black boxes which may contain more than one cooperating services.

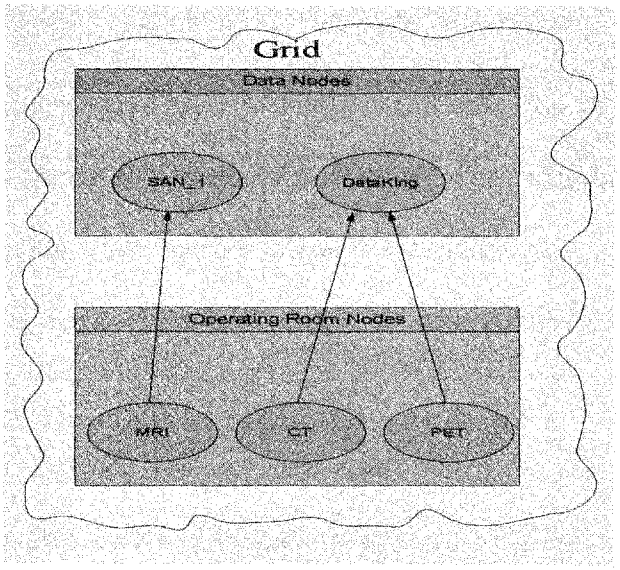

Illustration 4: Medical grid infrastructure

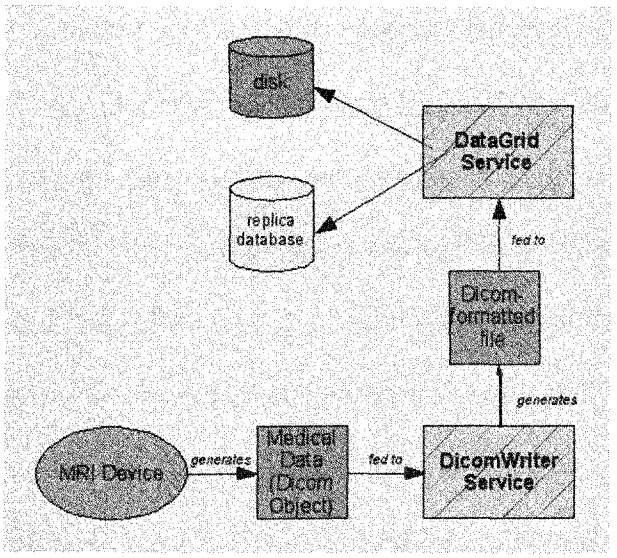

Illustration 5: Project work-flow

3.4.1 DataGrid Service

The *DataGrid* service was designed as a high-level service that uses two existing Globus services, the RFT and the RLS, to provide users with transparent access to grid storage resources. The main resource in a data grid is storage; specifically, storage volumes (such as hard disk partitions.) To represent individual storage volumes across the grid, the following resource properties were defined for the *DataGridService*:

- FreeSpace – the amount of space available in the volume.
- VolumeName – the physical name of the volume. For example, /dev/sda1 or C:\
- NickName – an easy-to-remember name for the volume.
- HostName – the name of the host where this data volume physically resides (i.e. the data node).
- Location – the physical location (area code) where the host resides. It could be used when deciding the best data node to retrieve a file from based on proximity. This is more accurate than the IP address, although it is subject to human error.

All of the resources, except for the volume name, are aggregated to the GT4 Index Service. Index Service aggregation is done by creating an XML-formatted file, called a registration file, and associating it with the service. The binding between the registration file and the service is specified in the Resource Home of the service, which will be explained in more detail later. The registration file specifies the resource properties to be aggregated, to what service they belong, and how often they should be refreshed, among other things. The volume name is not aggregated since it is system-specific, and of no use to grid members; since they merely want to store data, they are not concerned with what volume it is stored in. But it is still kept in the resource.

Illustration 6 depicts the use case of registering a data volume onto the Grid, which consists of the following flow of events:

1. Client tells broker she wants to register her hard disk drive into the Grid
2. Broker tells Factory that the client wishes to register a *DataVolume*
3. Factory tells Service it needs to access the ResourceHome to create the resource
4. Service gives Factory reference to the Resource Home
5. Service gives Broker the Endpoint-reference for the new resource (i.e. volume)
6. Broker tells client that the request has been processed

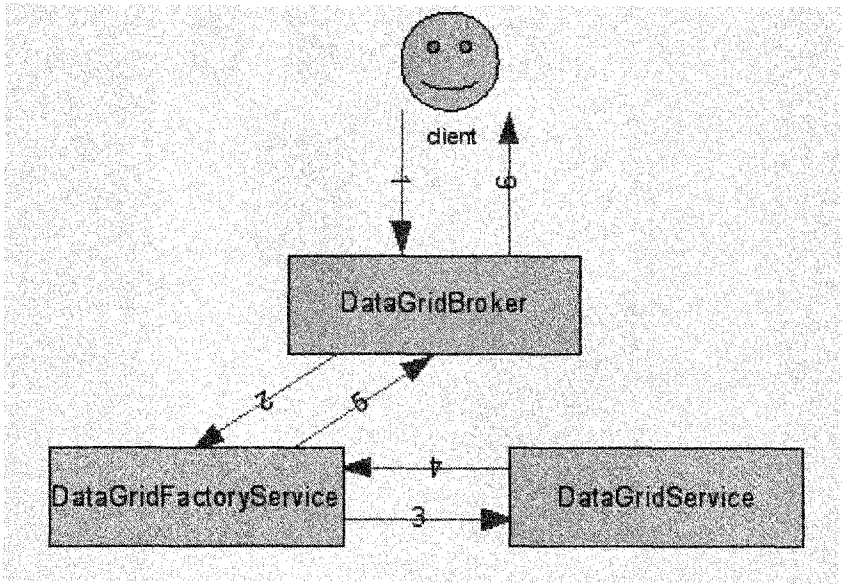

Illustration 6: Data volume registration use case scenario

The ultimate goal for data access on a grid is to allow seamless access to the data that is stored in it. That is, make the data grid seem like one huge disk drive. This service achieves this goal by providing typical file operations as public, service-accessible methods. On the client-side, this seems like a regular (local) function. One of the benefits of

service-oriented architectures is demonstrated here: all of the complicated tasks, such as finding and interacting with lower-level services, is negotiated between machines, and takes place "behind the scenes." The RLS helps make this possible by making physical locations abstract to the user. The user only needs to know the logical name, which is the same as if they were working with a local file system. This is shown visually in Illustration 7: the user just sees the big virtual disk (the larger cylinder with the hatches). In actuality, there are several disks, spanning various organizations.

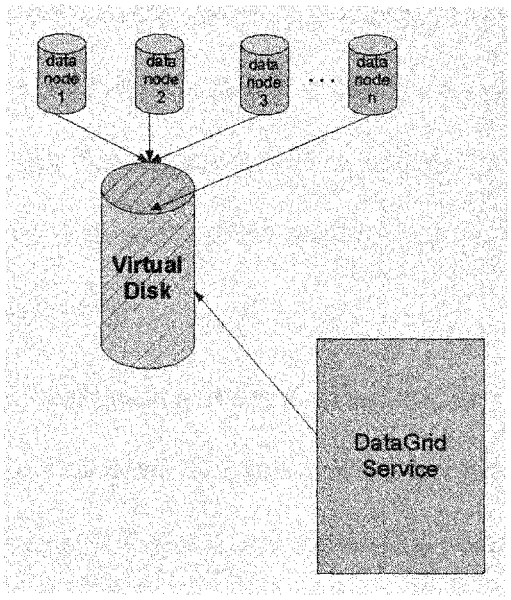

Illustration 7: Storage Virtualization

When designing GT4 services, Globus provides flexibility by offering several interfaces for programming all kinds of services, from simple services with no resources that can be deployed in a few minutes to very complex services to hold several persistent resources that require days of design and planning.

The basic design paradigm used for stateful Grid Services is the factory/instance paradigm. In this paradigm, the services are separated into a factory service and an instance service. Typically, the factory service is invoked when creating a resource. Then, a user invokes methods from the instance service on created resources. The idea is to separate the code that creates the resource from the code that manipulates the resources. This results in more manageable code. Also, the services can be further spread out across different systems. In some cases in may be favorable to have the factory and instance services in separate locations. Furthermore, if the instance service is particularly compute-intensive, it can be deployed in several servers without also having to deploy the factory service. The factory service can just be deployed in one location and call services in other locations. The benefits of this design pattern will be revealed when the services are described. For simple services with single or no resources, the creation and manipulation code is all placed in the same class.

For the *DataGridService*, where multiple resources (i.e. storage volumes) are utilized, the factory/instance pattern was first used. The volumes were registered through the factory service and stored as resources in the resource home. This resulted in an acceptable, albeit cumbersome code base. The service class started to get bulky and confusing because it was making too many decisions, such as which data volume to use from all the entries found, where to save data, etc. With SOA, it is best to have separate services whenever possible.

As a result of this, a broker was introduced to do some preprocessing of the requests. The Broker is actually an independent service. It delegates the task of actually transferring a file to the *DataGridService* after it does its processing. For example, when a client wants

to insert a file, the request will go through the Broker, who will discover one or more resources (data volumes) in which the data fits, determine the best one to insert into, attempt to create the logical file names (ensuring that they don't already exist) and create and send the *DataGridService* request.

The inside of the "black box" referenced in Illustration 8 portrays what services get called when the client invokes the service: the client does not interact directly with the services, he/she invokes the *DataGrid Broker*, who, depending on what type of request the user is making (i.e. register a volume, insert a file, etc.), invokes the appropriate call to the factory or instance service. For example, the scenario for a successful file input would go as follows:

1. Client to Broker: "I would like to input this file"
2. Broker to Factory: "I need a resource for storing X MB of data"
3. Factory to Broker "Here is the EPR for a suitable storage volume request."
4. Broker to Service: "I need to perform *inputFile*() on this EPR."
5. Service to broker: "Success."
6. Broker to Client: "Your request has been processed."

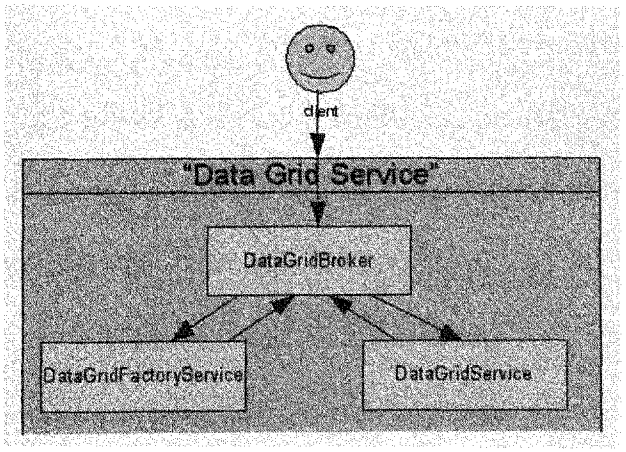

Illustration 8: Interaction between client and services of the *DataGridService*

Several challenges were faced while implementing this service. Interacting with GT4 services is fairly complex, it entails interacting with web services and familiarization with the Globus public interfaces and even its source code, in some cases. The biggest initial challenge faced when implementing this service was figuring out how to work with the RFT service API, mainly because of the need to pass a credential onto the service. Some research had to be done in order to find out what occurs at the GSI layer when invoking a service. What was determined was, when a user invokes a service, said user is identified as the "invocation subject." After verifying that the user has a valid proxy available, the service begins to execute. However, by default, the service is run as the user who owns the service, which is the user that owns the container (i.e. the globus user). Since this user is only used for working with the container, its account has no security credentials, so the transfer fails. Using what Globus calls a client security descriptor, the client can choose different forms of authorization and authentication to use. They can also decide if they want to delegate their credentials to the service, so that the service runs with their own identity, rather than that of the globus user.

In order to properly delegate credentials, the following changes had to be done to the code. First of all, the service's security descriptor had to be re-configured to run as the caller (as opposed to the host, which is the default). This is necessary because the RFT service, via its service security descriptor, requires a user's credentials to be used; it is not possible to use host credentials with this RFT service. The RFT service needs to authenticate the user (which can be done using the *MyProxy* API) and authorize the user by comparing the invocation subject (i.e. the user's delegated credential) to the list of authorized users of the grid. This presents a problem. The problem is that authenticating

with *MyProxy* requires the user's *MyProxy* password to be passed into the function. Requiring the user to do this is tedious and redundant (since they would have already inserted their password to get access to the Grid services in the first place.) Furthermore, such a scheme would require the user to provide a password for each transfer performed!

As a result, another method of obtaining the credentials to perform the transfer needed to be devised. One solution to this problem would have been to store the user's password in a text file. The problem with this is that the text file would need to be read by the globus user, which would be a security risk. An even better solution was found using classes available from the Globus API, namely, the *JaasSubject* class. This class works with the client security descriptor to perform delegation. Since the identity of the active user (i.e. caller) is being used, there is no need to ask for a specific user name and password combination. Another option would have been to use the Globus Delegation Service, which would allow greater flexibility, but that would have complicated the code even more. Since the added flexibility wasn't necessary, the *JaasSubject* solution was used.

With data transfer functionality working, the logical mapping using the RLS had to be implemented. Most of the necessary methods and classes were found while performing the RLS tests described in the Section 3.3. However, since the API interaction is now taking place at the container level, some security-related concepts must be addressed. First of all, the RLS client requires the host certificate and keys for the system hosting the RLS service. In order to make the implementation as flexible as possible, the code is set up to look in the file */grid/conf/rls.conf* for the host name, container key file, and container certificate file. This way, users can customize the system for their needs. However, if this

file is not present, it still looks in the default system locations for the credentials and attempts to create the connection to the local RLS using these credentials.

Error-handling and robustness is important for any application, but with networked applications, where so many things can go wrong and so many transactions are occurring, it is essential to have good mechanisms in place to handle, or at least give accurate information about problems. Therefore, considerable time was needed to go into designing the services for robustness. The transactions that occur in the *DataGridService* are particularly vulnerable to a few problems. For example, with the RLS, several things can go wrong. For one, the service may not be active (due to a crash, system restart, etc.), a given logical file name may already exist, etc. The Globus-provided *RLSException* class provides a mechanism for getting the error codes for errors that may occur while interacting with the RLS. Since a lot of the RLS code is low-level code implemented in C and accessed through Java's native interface (JNI), finding where these error codes were declared was not trivial.

Through browsing the source code, some useful information was found in the file *<globussrc>/replica/rls/client/library/globus_rls_client.h*. This source file explains how errors are projected and lists the error codes. Using this information, every time an error occurs with the RLS while invoking the *DataGrid* service, if there is a particular error that can be handled (based on the error code passed) and it is indeed that error that occurred, the proper handler code is called. Otherwise, the specific error is projected to the user. For example, if the user specifies a specific physical file to use, and that file exists, the user is notified of the problem. However, if the user did not choose a specific physical file name (recall that the logical file name is the only required name), and the service

generates an existing file name, it simply keeps trying different random file names until a non-duplicate value is generated.

3.4.2 DicomWriter Service

The *DataGrid* service provides an abstraction layer between a client's workstation and the Grid storage space. It also leaves room for higher level services to use its functionality to provide more application-specific functionality. This way, several layers of abstraction can be built. The next service being described, the *DicomWriter* service, will be used to demonstrate this. The *DicomWriter* service takes as input several parameters of user data from a medical study, such as images, patient's name, and type of study (MRI, CAT, etc.), converts it to a DICOM file, and uses the *DataGridService* to store the file. In addition, it provides a means of searching for DICOM files based on their attribute parameters.

The purpose of the *DicomWriterService* is to show how application-specific (in this case, medical files in DICOM format) functionality could be built on top of the *DataGridService*. The service may be used for regular client applications designed for human consumption and also for machine-based clients, such as MRI acquisition devices.

In its initial implementation, the service only has one public method, *inputDicom*. This method takes as input medical information and processes it, converts it to an actual DICOM file, and sends it to the *DataGridService*. No resource is used for the *DicomWriterService*, as its operation doesn't require state to be saved. Since there is no need for a resource, the service is based on the Singleton resource grid service, which is simpler to implement than the factory/instance scheme used by the *DataGrid* service. Rather than

having to go through a factory, resources can be directly retrieved from the resource home.

An object was created, named *DicomType*, which acts as a structure for storing various DICOM parameters. The parameters are used when creating the DICOM file to store the DICOM meta information. These parameters are sent to the inputDicom method, which is the main method of the service.

To aid in the process of creating DICOM-formatted files, the *Pixelmed* [33] DICOM API for Java was used. *Pixelmed* provides tools for generating DICOM files from patient-related meta-data. The DICOM specification gives a specific location in the binary file for the (2000+) DICOM attributes. *Pixelmed* creates the file by binding each attribute's location in the file to a constant. For example, the PatientName tag belongs at location 0x0010, 0x0010. When setting the patient's name in the service, an Attribute object is created with this location bound to it. The name is then added to this Attribute. When the DICOM file is created, it iterates through the list of Attributes and places them in the file according to their designated location.

The last thing that the *DicomWriter* must do is send the file(s) to a storage location using the *DataGridService*. This requires locating a *DataGridService* provider with sufficient space for storing the file, creating an endpoint reference to that resource, and delegating credentials to perform the transfer on the caller's behalf.

As with any service that uses external Java packages, the *Pixelmed* jars must be placed in the "lib" subdirectory of the directory where Globus is installed. Otherwise the container will not be able to find them at runtime, even if they are in the system classpath.

3.5 Improving the Persistence

So far, the described design has proven that Web Services, along with some Globus-toolkit provided software tools, do a good job of enabling computers to share their storage resources, forming a potentially huge storage space for inserting arbitrary data; additional services can be created for dealing with more specific data. In this case, medical data in the form of DICOM files is being dealt with. However, there is more that needs to be done in order to enhance collaboration. For one thing, the platform lacks the ability to query data that has been stored based on meta-data (i.e. DICOM attributes). In the case of DICOM files, which are the focus of this thesis, the attributes of medical studies should be available for querying. Without this feature, users are only able to access specific files for which they know the virtual file name. For example, if a researcher wants to look for trends in MR images for children less than five years of age, he/she will need to query the data grid for all MRI recordings matching this age criteria. Since this type of query is dependent on the type of file that is being dealt with, the logic for it is placed in services above the *DataGrid* service (since the *DataGridService* stores arbitrary files with arbitrary meta-data). In the case of this thesis work, this would be the *DicomWriter* service.

The use of separate services for different file types is somewhat similar to what *AliEn* does with their file catalog. They use separate tables for each directory. The difference is that with this scheme, the whole database is different, not just the table. Also, in this case, things are even more transparent since all the file names are generated automatically, the user need not worry about where the file was stored (i.e. host, path name, and base

name), only the unique Service Object Pair (SOP) Instance UID for the study (which corresponds to the logical file name), in the case of DICOM files. For other file types, a similar naming scheme can be used.

In order for the data to be query-able, there needed to be some kind of meta-information storage implementation. There were a couple of options here, each with its own benefits and drawbacks. The first is to use the GT4 Index Service, which is structured like an XML database; the second option is to use a relational database such as *MySQL*. The advantages and disadvantages of each were analyzed in order to decide the best approach to take.

An XML database is simply a collection of data arranged in XML-format. The following stub shows an example of possible XML data for medical records.

```
<medicalStudy>
  <date>08062006</date>
 <patientName>Hal E. Luya</patientName>
 <modality>MRI</modality>
 [...]
</medicalStudy>
<medicalStudy>
 <date>08112006</date>
 <name>Dan D. Lyon</name>
 [...]
</medicalStudy>
[...]
```

XML-based databases have several advantages. For one, they are easily-readable even by general audiences. Also, the files themselves are simple text files, so no special software is required to read them. What makes them most attractive in this project is the fact that the Globus Monitoring and Discovery Service (MDS) uses XML as its format of choice for storing information. Thus, an XML database is automatically generated by simply

registering the parameters for each registered study as resource properties. In this case, the resource being acted on (i.e. queried, stored, etc.) would be an XML-representation of a DICOM file stored in the data grid. There are, however, some drawbacks to this approach. For one thing, the performance will not be as good as having a relational database. However, this may not have a significant effect since the main bottleneck in a Grid scenario will probably be the network latency. Also, *XPath*, the query language for XML databases, is not as mature as the Structured Query Language (SQL), the query language for relational databases.

Another drawback to XML is, since XML files are simple text files (with special formatting instructions), they require parsing. Simply reading the entire text file involves linear time complexity; this however is not a huge issue since modern systems perform this very quickly. Also, there are XML parsers, such as SAX, that don't need to read the entire file at once to do their parsing. These considerations are important because although this particular application's data is very simple in structure, as can be seen from the above stub, there will be a lot of data, which must be taken into consideration.

Relational databases, on the other hand, store data in binary format, which is faster. There is a bit of overhead involved in connecting to the database server, but it should still be a lot faster than querying an XML database. As one might expect, a relational database scheme would be considerably more complex than using the MDS registry. For one thing, grids are meant to be decentralized (similar to peer-to-peer networks). This means that accessing the database would actually be a matter of accessing a service that communicates with the databases distributed throughout the grid. Furthermore, for reliability it would be necessary to replicate that database, thus increasing the level of complexity.

Query times must also be taken into consideration. Modifications would take even longer in an XML database, since the text file has to be regenerated every time something is changed and/or added. In a relational database, the process is simply a matter of updating a record in a table, a very simple operation with negligible cost. The only overhead in the case of the distributed relational database, then, would be the cost of interacting with remote services.

Despite all of this theoretical speculation, it is still difficult to decide what the best option is. Since this is a critical decision in the design, both types of services were implemented and benchmarked in order to determine the best solution for the job.

3.5.1 Implementation of the Services

3.5.1.1 XML-Based Storage

As mentioned previously, an XML database can be constructed by storing all of the meta-information about files as resource properties of a service. The logical service to put these properties was the *DicomWriter* service. The *DicomWriter* service is generic enough to be used as the holder of these resource properties. After all, its purpose is to insert and query DICOM files to and from the Data Grid. However, some overhauling of the implementation design of the *DicomWriter* service was necessary in order for it to be able to store this information. In its original design, the *DicomWriter* service simply created DICOM files from generic patient information. There wasn't any need for resources/state information. The WSRF service implementation flexibility allows such services to be implemented easier than services with resources, let alone multiple resources.

As a result, several changes had to be made to the design of the *DicomWriter* service. For starters, it could no longer be a singleton service; it needed to have the ability to store multiple resources. To achieve this, it was separated into a factory service and an instance service, similar to the *DataGrid* service. Unlike the *DataGrid* service, a broker wasn't necessary.

When implementing the new service, only the algorithm for handling the input of the file was retained. The first step was to create an additional Web Services Description Language (WSDL) file for the factory service. Since the resources are now files, the factory service provides the method for inserting files (and subsequently registering them into the index). The instance service provides the methods for querying and updating files. As such, these are declared in their respective WSDL files. The resource properties added to the service for testing purposes are: Uuid (the SOP Instance Uid, prefixed with "U" to signify uniqueness), StudyDate, Modality, PatientName, PatientId, BodyPart (the body part examined), PatientSex, and PatientAge.

The *DicomWriterNamespaces* class was replaced by a *DicomWriterConstants* class, which declares the resource properties for the *DicomWriter* namespace. Also, a *DicomWriter*Configuration class is created to allow the resource class to access service information stored in the deployment files, which also had to be changed. The deployment information for the factory service had to be added to the deployment file (*deploy-server.wsdd*). Also, the resource bindings need to be provided in the JNDI configuration. In this file, the *DicomWriter* resource is bound to the *DicomWriter* ResourceHome, which is bound to the instance service (*DicomWriterService*). Also, the instance service is separately bound to the factory service.

The new *DicomWriter*Factory service resembles the implementation design of the old *DicomWriterService*, since it is the provider of the *inputFile* method. The difference is that, in addition to transferring the file, it also extracts its meta-information and publishes it to the Index Service.

The *DicomWriterService* became a totally new class with two public methods: *updateFile* and *queryFile*. The *queryFile* method accesses the index service to locate files matching a specified criterion. Since *XPath* is the language of choice for making queries to the index service, the *queryFile* method takes as input an *XPath* query string. *XPath* enables searching of an XML document using file-system-like hierarchal searching through the logical structure of the document. The *XPath* string can be constructed on the client side from input given by the user, for example using the input of a form.

It became apparent while developing an algorithm for querying through the index service that a separate (helper) class should be used for interacting with the index service. A lot of code is identical, regardless of the type of resource properties, when working with the index. For example, much of the code for accessing the files is literally the same as that of accessing the data volumes (in the *DataGrid* service). Therefore, adding a new class was the ideal solution, as it provided a way of reusing the same code and also making the service code easier to understand.

The newly added class to the project was named *IndexServiceQuery*. The class is instantiated with an *XPath* string and provides one method to its callers: *executeQuery*. The *executeQuery* method queries the default index service for matching elements according to the *XPath* string. It returns an array of *MessageElement* objects, containing all of the

matches. *MessageElement* is the base type of nodes of a SOAP message's DOM tree [34].

After creating the *IndexServiceQuery* class, there still seemed to be some tedious type conversions necessary to convert the returned results (i.e. *MessageElements*). For one, the *MessageElements* need to be converted to *EntryTypes*. The entire *EntryType* is not needed, since the *EntryType* contains all information about the resource, not just the (aggregated) resource property values. For example, an *Entry* from the *IndexService* may look like this:

1. <ns13:Entry>
2. .....
3. .....
4. <ns11:AggregatorConfig>
5. <ns11:GetMultipleResourcePropertiesPollType>
6. <ns11:ResourcePropertyNames>dcm:Uuid</ns11:ResourcePropertyNames>
7. <ns11:ResourcePropertyNames>dcm:StudyDate</ns11:ResourcePropertyNames>
8. <ns11:ResourcePropertyNames>dcm:PatientAge</ns11:ResourcePropertyNames>
9. .....
10. </ns11:GetMultipleResourcePropertiesPollType>
11. </ns11:AggregatorConfig>
12. <ns11:AggregatorData>
13. <ns1:Uuid

    mlns:ns1="http://www.globus.org/namespaces/*DicomWriter*">1.3.6.1.4.1.5962.99.

    1.3340495864.280305340.1158686731256.1.0</ns1:Uuid>

14.<ns2:StudyDate

xmlns:ns2="http://www.globus.org/namespaces/*DicomWriter*">4/4/2000

</ns2:StudyDate>

15.<ns3:PatientAge xmlns:ns3="http://www.globus.org/namespaces/*DicomWriter*"

16.>36</ns3:PatientAge>

17....

18.</ns11:AggregatorData>

19.</ns1:Content>

20.</ns13:Entry>

Only the data on lines 12-17 (i.e. the aggregator data) is useful. What this means from a coding standpoint is that after getting the *Entry*, i.e. the *EntryType*, from the *Message-Conent*, the aggregator data needs to be extracted. At this point, an XML representation of each result is available. Finally, the XML has to be parsed to extract the parameters and create the appropriate object (i.e. *DicomType* or *DataVolume*) out of the extracted values. Although this code is specific to each service's resource properties, putting it in the same service class would lead to a lot of clutter, and is therefore best left as a separate class that extends the *IndexServiceQuery* class.

Actually, an even better approach, since the *EntryType* data is not particularly needed, nor is the raw *Aggregator* content, is to do all these type conversions in the *IndexServiceQuery* class. Furthermore, some of the XML parsing can also be done in this class. This way, the only functionality needed in the higher-level classes is conversion to *DicomType*, *DataVolume*, etc.

Since the order in which the resource properties are set in the aggregator data is constant, the values of each of the properties can be retrieved by their index, which is the order in which they are inserted in the registration file. For example, in the *DataVolumeQuery* class, the nickname, location, hostname, freespace, and data directory resources are stored and received as items 0,1,2,3, and 4, respectively. If this weren't the case, the aggregator data would have to be parsed to ensure that the proper value is assigned to each resource property. Some sanity checking is still done to ensure that the correct values are being assigned to each property.

At the client side, all that needs to be done is create an *XPath* string and send it as a parameter to the *queryFile* method of the *DicomWriterService*. The actual query string could be generated based on form entries, so that the user doesn't need to know *XPath* just to make a query. This is also a good security measure, since it is not wise to let users insert arbitrary XPath queries.

3.5.1.2 Database Service Approach

A distributed database would presumably be faster at the expense of having a more complicated implementation and less integration with the Globus Index service. Not only would meta-data be accessible directly from the index, but there would also be a need to have a separate table for file permissions/authorization as well as certain private meta-data. With the XML/*IndexService* approach, this could be done by means of resource-level security. With resource-level security, each resource created (in this case the record inserted into the index) is only accessible by the creator of the user, based on the grid identity that they are running as.

A naive approach at this would be to query all servers with the database service for entries matching a particular criterion. This would make replication particularly difficult though. In typical production distributed database scenarios, separate databases servers are connected together by means of database links; a database link allows a local database to access data on a remote database server as if it were on the same (local) database server. Location transparency can be achieved by creating "synonyms" – easy to remember aliases for remote database tables. This way, the remote database user, name, and/or location don't have to be specified for each query/update. However, this can become a problem when dealing with so many distributed administrative domains and concurrent operations.

In a Grid scenario, it seems more appropriate to have a single database, perhaps replicated across several domains for faster access and reliability. The replication would add complexity as it would be necessary to create a lock on the database file while transferring its contents, but that won't be looked at for now for the purpose of benchmarking. To demonstrate such a scenario, a basic database service was created and deployed on a single node.

The database service was implemented as a WSRF service, appropriately named *DatabaseService*, which will be accessible remotely. A separate class, *SimpleDB*, was created to interact with the service to access a local database. When a client accesses the service, the service uses the request to build an SQL statement and sends it to the *SimpleDB* class, which connects through Java Database Connectivity (JDBC) with the local database. Of course, a database also had to be created at the node providing the service. Since *MySQL* was already being used for the RFT and RLS databases, it was also used for this database,

which was named *dgsDatabase*. The database only needed one table, with the following attributes (the same as the resource properties used in the XML implementation, plus the logical file name): Uuid, StudyDate, Modality, PatientName, PatientId, PatientSex, BodyPart.

The *SimpleDB* class provides some basic methods for connecting to, querying, adding, and updating entries from and into a local database. In this class, the database name and connection parameters are arbitrary, but the *add* and *query* methods are specifically for returning *DicomType* objects. The update method is also generic, although this makes updating more complicated on the service side, since individual parameters need to be checked to see what parameters of the file must changed; therefore the attributes must then be changed in the database to reflect this.

The *DatabaseService* provides the remotely-accessible query, insert and update methods. For queries, it takes as input a *DicomType* populated with the desired parameters for the query. The *queryDatabase* method looks for all non-null parameters from this *DicomType* and creates an SQL SELECT statement based on these parameters. This statement is then passed to the *SimpleDB*'s *readRecord* method, which connects to the database and performs the query.

The insert method also takes as input a *DicomType*, inserts the file using the *DataGrid* service (the same way the *DicomWriterService* does) and then extracts the parameters of the *DicomType* and creates an entry in the local database with these parameters.

The update method takes as input the SOP Instance UID of the study and a *DicomType* with the updated values to modify. It then attempts to overwrite the existing file, using

the RFT service. There is no need to use the *DataGridService* when updating since the logical name and mapping to the physical file already exists. Once the transfer has successfully occurred, the necessary database entry values are updated.

3.5.2 Performance Evaluation

3.5.2.1 Methodology

To evaluate the performance trade-off of using the GT4 Index service for queries, updates, and insertions, as opposed to using a relational database, a set of inputs was generated and performance was evaluated for the two services. The following benchmarks were performed to compare the performance of the two approaches. For insertion, three data sets were created, with 50, 250, and 500 randomly generated sets of DICOM attributes. The following methodology was used for generating the data. The SOP Instance UID was generated as a unique random integer, 36 characters in length. A list of current entries was kept in order to ensure that no duplicates were entered. This is not as complicated as actually generating the SOP Instance UID the same way set by the DICOM standard, since such a complication isn't necessary since compute time for that is negligible: SOP Instance UID generation using *Pixelmed* only takes approximately 0.2ms on the head node of the cluster. The modality was randomly chosen as either "MR," "PET," or "CT." The patient names were generated as random strings with a maximum length of 20 characters for first name and last name. Patient sex was randomly chosen as either "M" or "F." The Study ID was chosen as a random, unique number of length 7. Age was chosen as a random number, with a maximum of 88. The study date and body part attributes were simply hard-coded, since they won't be used for querying. The program outputs all

of the entries into a comma-separated-variable (CSV) file. This file is then read by client-side programs which perform insertions into the Index Service or database for the *DicomWriter* and *DatabaseService*, respectively.

To test query performance, a few different queries were evaluated. The first query performed searches for all MR files, i.e. all studies in which the modality equals MR. The next three queries search for patients by name. The first one looks for the first name in the list of entries (which would be the first entry in the database or index). The next one searches for the middle entry (i.e. 25 for the list of 50, 250 for the list of 500.) The last one looks for the last entry in the list. The next three queries are based on age. They look for entries where the patient's age is equal to, greater than, and less than 25, respectively. The final query is the only one that looks for multiple attributes. It looks for all entries in which the modality is CT and the patient is older than 20. All statistics are tabulated in Table 1.

Table 1: Data Input Times for *DicomWriterService* and *DatabaseService*

Number of Entries	DicomWriterService (sec)	DatabaseService (sec)
50	3.750	3.838
250	4.615	3.588
500	N/A	2.355

Before even being able to obtain all the results, problems were encountered with the *DicomWriter* service. With fifty entries, it ran fine. However, when attempting to insert more than 300 entries, problems began to occur. First, the container started throwing "Out of Memory" exceptions, even after setting the maximum heap size to 1024MB

(which is half of the total system memory!) Since using this much memory can impact system performance, and because this data would have needed to be persistent anyway, the *DicomWriter*Resource class was modified to make its resources persistent, i.e. placed in secondary storage rather than kept in memory. Unfortunately, doing so will cause even more of a performance penalty. The Globus API does allow resources to be cached based on how often they are accessed, or whatever other scheme a developer wants to create, but that is not helpful in this case since all resources must be read when querying data, so there is no useful caching scheme.

Adding persistence requires a few modifications to the *Resource* class. The Globus API does a lot of the lower-level I/O work, but there are still a few modifications needed. First of all, since resources are now being kept in disk, they must be loaded from the disk[2] in order to be read. As a result, a *load* method had to be added to the *DicomWriterResource* class. This method validates the resource key and initializes the resource using the resource properties it reads from disk. The resource properties are read from and written to disk using Java *ObjectOutputStream* and *ObjectInputStream* classes, which help serialize object data so that it can be written to and read from secondary storage.

The other method that must be implemented is one to store resources onto secondary storage, hence it is called *store*. The store method does the opposite of the load method in that it serializes the resource data and stores it to disk using an *ObjectOutputStream*. This method is called anytime resources are modified, i.e. when initializing the resource or when setting one of the resource properties. All files are placed in a specified directory

2. Actually, they need not necessarily be in a disk, any kind of secondary storage will do. But disks were the chosen medium in this case

within the globus user's home directory. Actually, to ensure no files get overwritten prematurely, the entire process is first performed on a temporary file; if no problems occur during execution, the temporary file is copied to this location as a resource file. If a problem occurs, which is quite possible with I/O code, especially when being done as a remote transaction, the temporary file is simply deleted, and the resource is not saved. If it was an existing resource, the original is left untouched. Of course, the user is told the outcome in either case.

The execution times actually turned out to be fairly similar for the two services. The total elapsed time for *DicomWriterService* was 194 seconds and the *DatabaseService* 187 seconds. The *DicomWriterService* averages just 0.14 seconds slower per insertion (194 vs. 187) when performing 50 insertions. The time for 500 insertions wasn't taken into account for the *DicomWriterService* since problems arose, although it did finish doing its job. The *DatabaseService* completed it in 499 seconds, which is approximately 1 insertion/second, which is a very good average. Note that the database service created for this test was more simplified than it would have to be for a fully-functional implementation. For example, there would be a need for an additional lookup into a permissions database to ensure a file can be read or updated by the user attempting to do so.

One thing that was noticed while doing these evaluations is that the first insertion always takes the longest, regardless of what service is being used. This is due to the message passing that occurs in negotiating the remote connection, obtaining the port type, etc. This raises an interesting question to be considered when designing the client software, how will things occur in a practical scenario? That is, will a client maintain a connection that is constantly on, and with a port type that is indefinitely instantiated (until said cli-

ent's credentials expire), or will a new port type be instantiated on every insertion? How many insertions are typically done at a time? These and other questions will be looked into when designing the client side applications.

To get a better idea of how long each transfer (just the transfer itself) takes, the tests were repeated; this time, the time for each individual insertion was measured. This showed that the first insertion took about 10 seconds while the rest ranged from 3.5 to 6 seconds. Therefore, to get a more accurate measure, the average of all the insertions after the first one was then calculated and tabulated. Note that the container was restarted for each test and the values for the *DicomWriterService* were measured using in-memory (i.e. non-persistent) resource storage.

3.5.2.2 Benchmarks

The same observation about the first transfer taking the longest was also found to be true when querying. Therefore, the programs for querying were also modified to first search for a knowingly non-matching query. Note that the same port type is used for all queries, which explains why all the queries after the first one go a lot faster. The results are tabulated in Table 2.

From the results obtained, it can be seen that the *DatabaseService* has a slight edge in performance for insertions and queries, in almost all cases. Also, it became clear that using the *IndexService* to store meta-data was not going to be neither reliable (due to the container crashing issues) nor efficient. The *DatabaseService* provided good performance, reliability, and scalability. However, as mentioned earlier, the implementation used for the tests is very basic: it could only handle basic queries and insertions and it is only

useful in applications where there is only one instance of the database available. This is not a good scenario in a distributed system, where a single point of failure (i.e. a faulty system) can cause every node that relies on the database to be become unusable until said system is brought back up.

Table 2: Data Query Times for *DicomWriterService* and *DatabaseService*

Criteria	DWS_50	DWS_250	DBS_50	DBS_250	DBS_500
No Match	4106	4851	3605	3842	6667
MR Files	719	1783	729	810	1891
Name (first in list)	421	748	294	248	1554
Name (middle of list)	419	611	237	219	1207
Name (last in list)	458	613	186	244	1359
Age = 25	400	662	191	278	1239
Age less than 25	419	1541	252	467	1052
Age less than 25	750	3317	618	1717	1585
Age less than 20 and CT	365	2274	169	263	1281

3.5.3 Solution: OGSA-DAI

Since implementing a distributed, Grid-optimized, reliable, and robust database system would require far more research and design time than are currently available; some further research was done regarding existing distributed database implementations that could be used in a Grid. The ideal solution for the job turned out to be OGSA-DAI, a UK-based project, where DAI stands for Database Access and Integration [35]. OGSA-DAI provides middleware for exposing data resources on a Grid. It supports various data resources, such as relational databases, XML databases, and flat files. Furthermore, it provides seamless access to these resources – users need not know how the data resource is

implemented, they just perform data transactions using whatever methodology they prefer, such as SQL, *XPath*, etc.

Using a technology that has a solid development team, complies to standards, and is well-supported and adapting it to the work being implemented in this thesis is superior to having to do any implementation that cannot possibly be as robust and complete as a proven technology, given the time constraints of thesis work. OGSA-DAI is being used by several organizations world-wide, their website alone lists 37 major projects. Of course, the performance of OGSA-DAI has to be compared to the other two services to see how it compares; therefore, after setting it up, the tests will be re-run.

Not implementing an application-specific database scheme comes at the expense of having to do some additional configuration of third-party software. A few issues were encountered while configuring OGSA-DAI, but the basic process is outlined here. Note that since the database used for the *DatabaseService* worked well, it was used with the OGSA-DAI-based service also, so no additional databases were created.

The following procedure worked successfully for getting everything working under version 4.0.3 of Globus. First, a grid-archive file for the service needed to be created and deployed onto the container. Then, since OGSA-DAI requires users to deploy individual resources, such as data service resources, such a resource had to be deployed. In this case, a *MySQL* resource had to be created. Then, the data service resource needs to be deployed. This required creating a configuration file with all the parameters of the data source, in this case, the *MySQL* database (*dgsDatabase*). This file lists the name of the

database, how to access it (in this case using the JDBC connector for *MySQL*) and the credentials to use, among other things.

After everything was configured, some searching had to be done through the documentation and API for OGSA-DAI to see how to interact with the database. In order to perform the necessary functions, which include inserting and updating information to the database and querying information from it, the following classes proved useful:

- *uk.org.ogsadai.client.toolkit.activity.sql.SQLUpdate* – for inserting and updating. This is also useful for deleting entries. Basically, this class covers all *execution* SQL commands
- *uk.org.ogsadai.client.toolkit.activity.sql.SQLQuery* – for performing queries, i.e. performing SQL SELECT statements.
- *uk.org.ogsadai.client.toolkit.security.wsrf.SecurityConfigProperty* – Used for setting security parameters.
- *uk.org.ogsadai.client.CoGUtil* – Since the container is running on the secure HTTP protocol, a method from this class had to be called in order to run OGSA-DAI clients.
- *uk.org.ogsadai.client.toolkit.activity.sql.WebRowSet* – object for returning results from a query or execution. The results are returned as an XML *WebRowSet* [3]
- *uk.org.ogsadai.client.toolkit.activity.ActivityRequest* – OGSA-DAI uses the concept of "activities" for interacting with data sources. Since many times several ac-

3. Note that one is an XML *WebRowSet* and the other is an OGSA-DAI *WebRowSet*

tions are performed on the data sources, these *ActivityRequest* objects are used to encapsulate various activity requests. The requests can be performed on any Activity object (i.e. any subclass of *Activity*), which includes *SQLQuery*, *Xquery*, *WebRowSet* , CSV, etc. For example, when performing a query, the SQL command and the *WebRowSet* that is to contain the output are both added to the *ActivityRequest* before performing the activity. After performing the request by calling the perform method of the Service class, the *WebRowSet* is populated with the matching queries.

- *uk.org.ogsadai.client.toolkit.Response* – this interface declares the necessary methods for the Response classes, such as *ActivityResponse*. Particularly useful are the methods for printing results as strings or in XML-Document-compliant format.

- *uk.org.ogsadai.client.toolkit.service.DataService* – this interface declares the necessary methods that the services provided by OGSA-DAI, such as *WSRFDataService*, implement. The most important one being the perform method, which invokes the data operation.

- *uk.org.ogsadai.client.toolkit.GenericServiceFetcher* – OGSA-DAI provides this class so that generic types can be used for interacting with data sources. For instance, instead of having to declare a *WSRFDataService*, *DataService* can be declared and the exact type of *DataService* can be extracted using the *GenericServiceFetcher's getDataService* method. Based on the URL passed to this method, it is able to determine the type of *DataService* from its WSDL document.

Rather than creating a client that directly interacts with the OGSA-DAI resource, a higher-level service was created. The service performs the necessary DICOM conversions (file to meta-data to database conversion for insertions and database-to-file conversion for queries). Having a separate service also allows encapsulation of the data source being used, and saves clients from having to deal with the intricacies of getting results and performing queries through the OGSA-DAI API. This is essentially what the *DatabaseService* does. Therefore, this same service was modified to work with OGSA-DAI, instead of using the *SimpleDB* class that it currently uses. So to keep things as similar as possible, a class with utility methods similar to those of *SimpleDB* was used. Since things were no longer very simple, the class was named *DicomDatabaseConnection*. The core functionality is much the same as the *SimpleDB*; in fact, very few changes were required to the *DatabaseService* class. However, the implementation now uses the above classes from the OGSA-DAI API to interact with the database, rather than using the JDBC connector directly, as it was with the earlier implementation.

Queries are still performed using SQL, even though OGSA-DAI allows various query languages (such as *XPath*) to be used regardless of the database implementation. They do this by encapsulating queries (as well as other things) into *Activity* objects (see above). SQL seemed like the best approach due to the relational nature of the data that is being stored. The results of the query are returned encapsulated in a *WebRowSet* , which is then converted into an array of *DicomType* containing the matching results and returned to the client.

Once the service was working, the statistics for insertion and data query were obtained, using the same criteria that was used for the other services. The insertion and query times are listed in Tables 3 and 4, respectively.

Table 3: Data Input Times for OGSA-DAI data service

Number of Entries	Average Data Input Time (s)
50	5.067
250	4.723
500	7.472

Table 4: Data Query Times - OGSA-DAI Data Service and Basic *DatabaseService*

Criteria	DBS_250	OGSA_250	DBS_500	OGSA_500
No Match	3842	5723	6667	10025
MR File	810	2347	1891	7995
Name (first in list)	248	1152	1554	5753
Name (middle of list)	219	1032	1207	5304
Name (last in list)	244	1078	1359	5361
Age = 20	278	1180	1239	5574
Age less than 20	467	1564	1052	6051
Age greater than 20	1717	3730	1585	9057
(Age< 20) & (Modality = CT)	263	1142	1281	5252

3.6 Parallel Correlation Integral Application for Epilepsy Detection

3.6.1 Background

The correlation integral has been shown to be an effective method for the detection of seizure onset (although its use in predicting them has been discounted) [18] [19]. Since

there is still no conclusive method for predicting the onset of epilepsy, and doing so is beyond the scope of this thesis, the correlation integral was decided on as the method of choice due to its validity for detection (which is still an important first step towards prediction) and the ability to parallelize the algorithm.

The correlation integral was originally used in the field of seizure prediction due to its sensitivity to non-linearity in signals. Non-linear signals have been shown to characterize the epileptic regions of the brain while a seizure is occurring [19], which makes the correlation integral a good choice for the task. Equation (1) describes the formula used for the correlation integral.

$$\sum_{t=1}^{N}\sum_{s=t+1}^{N} H\left(\left(r - \left\|v_t - v_s\right\|\right)\right) \tag{1}$$

Where N is the number of m-dimensional vectors, V_t and V_s are vector series, r is a predefined distance, and H the Heaviside step function. Since a time series is what is being read from the file, the signal must first be converted into a vector series using the formula in Equation (2), where m is the embedding dimension and τ is the time interval.

$$\vec{x}(i) = \left(u(i), u(i+\tau), \ldots, u(i+\tau(m-1))\right) \tag{2}$$

3.6.2 Implementation Details

The data to be analyzed is read from a text file. It is arranged in rows and columns, with the columns representing the signal voltage for each electrode placed on the scalp during the EEG at different instances in time. Since the program needed to be as efficient as possible to get the maximum performance from the computing resources, the programming

language of choice for writing the program was C. Actually, this C program was devised from a prototyped MATLAB implementation to ensure the algorithm produced correct results. Once the serial version was implemented and tested for validity, the parallel implementation was devised. This two-step process makes debugging easier, since bugs related to the computation algorithm itself could be fixed before fixing problems with making it parallel. It also allows the performance of the serial and parallel versions to be compared in order to test the scalability of the parallel implementation.

The main steps involved in implementing the correlation integral in software were:

1. Read the signal from file and arrange into a 2-dimensional array
2. Separate the signal into windows
3. Obtain a vector series from the signal window, using equation (2)
4. Get the correlation integral, using equation (1), and record the number of distances below a predetermined threshold for the current window and electrode

A window size of 500 samples was used. For obtaining the vector series, the embedding dimension (m) was set to 4 and τ to 3.

Once the serial implementation was working, it was necessary to design a parallel version. A couple of options existed for doing so. Since each electrode is evaluated independently of all the others, one option was to process each electrode (i.e. column in the file) using a separate processor, or to assign a subset of the electrodes to each processor, depending on how many processors are available. A data dependency graph for this algorithm is shown in Illustration 9. The other option was to process each window with a

separate processor, since the windows are also processed independently of each other. With the first option, the amount of distribution possible is more limited since there are always going to be more signal windows than electrodes, except with small data sets (which is certainly not the case here) or very large window sizes. For example, considering a data set with 88 electrodes and using a window size of 500, it would require a data set of just 500 x 88 = 44,000 lines (which is very little data) for the second method (distribute by window) to provide more parallelism. However, in order to distribute by window, it would be necessary to read the entire contents of the file before distributing the load, since knowing exactly how many windows to process allows better load balancing based on the number of processors available.

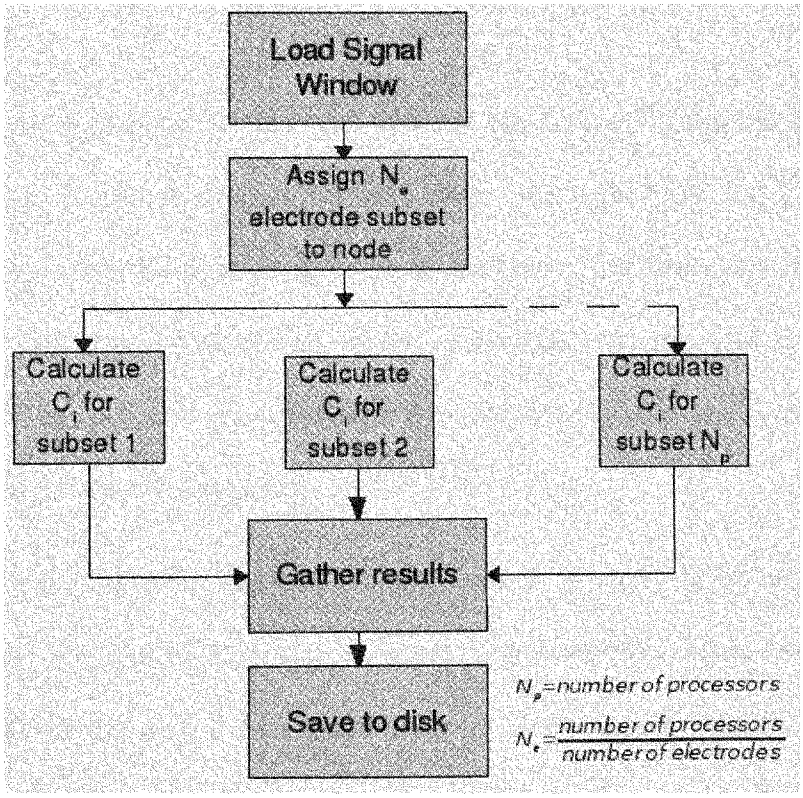

Illustration 9: Data Dependency Graph: Distribute electrodes

It was decided on to use the first method (distribute by electrodes) for a couple of reasons. For one thing, the computations for each individual electrode are relatively fast, at least for the sample data set. Another problem is the aforementioned problem of having to read the entire file before even starting the computation. A program was written to do just this for different file sizes to see the read times. The results can be seen in Table 5. The results show that too much time will be wasted reading the files beforehand to see how many lines they contain. However, since breaking down by electrode takes increasingly longer as the data set size increases, this approach may not be optimal either.

Table 5: Read times for signal data files of different sizes

File Size	Time (s)
500 MB	14.1
1 GB	39.0
3 GB	80.0

Another drawback to this approach is the fact that the width (i.e. the number of electrodes) of the data set is not likely to increase much. As such, there will come a point (i.e. *number of processors = number of electrodes*) where no performance will be gained by adding more processors. Still and all, it seemed like this method would perform better due to the lack of pre-processing needed, so it was attempted first and its results (in terms of computational efficiency) evaluated.

In order to make the algorithm parallel, a change had to be made to the order in which processing occurs. The serial algorithm traverses each electrode's signal value, one at a time, until it reaches the end of the current window. To do this in parallel, it would require the worker nodes to send their results after each window, and to know which window to do in the present iteration. As a result, the head node would need to do more or-

chestrating to tell the worker nodes what window to operate on. This would result in an unnecessary complication of the algorithm. Therefore, the algorithm was changed to do all windows at once for the subset of electrodes assigned to each processor and the results returned after they were all done. When the processor is done, it sends the results for the entire signal, which the head node can easily break down into separate windows since the window size is simply the quotient of the total number of rows in the file and the assigned window size.

Once the algorithms were tested and working, the performance was evaluated. There were three implementations in total: the MATLAB script, the serial version written in C, and the parallel version, which is an extension of the C version. The performance of each implementation is tabulated in Table 6.

Table 6: Computation times for Correlation Integral

Implementation	Data Set 1 (60,000 lines)	Data Set 2 (1,000,000 lines)
MATLAB (on workstation)	2658	N/A
Serial C (on workstation)	61.0	N/A
Serial C (on cluster)	39.16	N/A
Parallel C, 1 processor	40.32	21.25
Parallel C, 2 processors	22.82	16.79
Parallel C, 4 processors	14.56	13.85
Parallel C, 8 processors	11.32	12.71
Parallel C, 16 processors	11.07	12.72

Since a license for MATLAB was not available for the cluster, the performance of the MATLAB version was evaluated on a workstation with a 2.4GHz Intel Pentium processor and 512 Megabytes of main memory. The C implementation of the program was run on this system as well for comparison. The dataset used consists of 60,000 entries with 88 electrode values. To test the scalability of the parallel algorithm for increasing input size, another dataset was generated by duplicating random windows from the original data set. The second data set consists of 1,000,000 lines of signal data. The sizes of the data sets were 36 and 595 Megabytes for the smaller and larger sets, respectively.

From the results given in Table 6 and shown in Illustration 10, it can be seen that some improvements need to be made to the algorithm in order for it to scale better, i.e. take more advantage of having more processors available. The smaller data set scales well up to a maximum of 5-6 processors, but the benefits begin to decline rapidly after that. As the data set grows, scaling gets even worse. The table also shows that MATLAB would have not been an acceptable choice, unless the code was highly optimized.

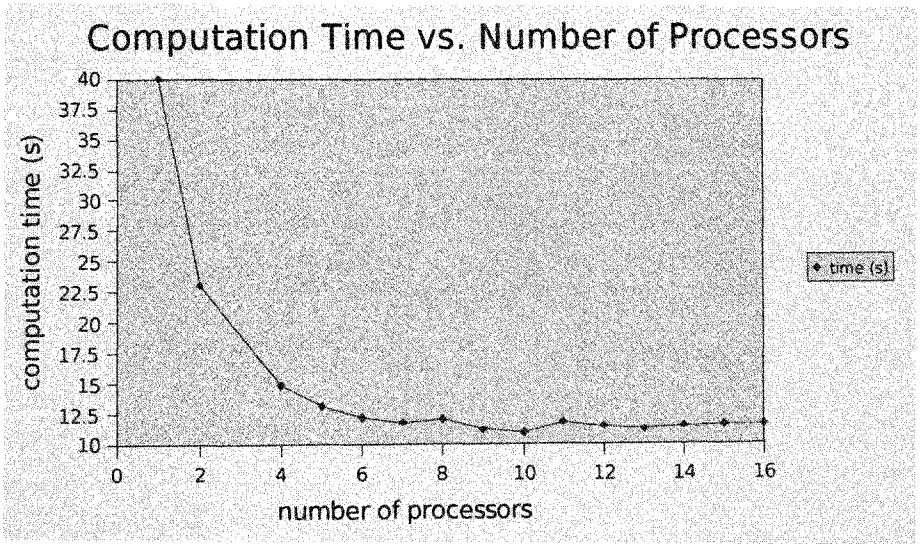

Illustration 10: Scalability of the algorithm

In order to help determine where the bottlenecks were in the code, the Multi-Programming Environment (MPE) was used. This is a toolset provided by MPICH for analyzing MPI programs. One problem that was noticed while using the MPE profiler is that the part of the code that reads the signal from the file always reads the entire signal, even though it only processes a certain percentage of it. This is because it needs to read the file on a line-by-line basis, even though only a subset of the columns (i.e. electrode values) is to be processed. As a result, this part of the code always takes equally long (about 4.5 seconds) no matter how many processors are used.

Another thing the profiler revealed is that the actual computations, i.e. obtaining the correlation integral only, actually scales well up until eight processors are being used. On the transition from 8 to 16 processors, compute time for the correlation integral decreases roughly 20 percent, which is not good considering that the processing power is being doubled. This means that the scalability problem involves the computation step as well as something else in the algorithm.

4 The Client Interface

Up until now, all the tests have been run using command line clients, which are obviously not acceptable for general audiences. Not only is a graphical user interface (GUI) more attractive and user-friendly, it can also display more information concurrently, such as Data Grid status information, which may include total amount of free space available, number of pending transfers, etc. All this can be displayed concurrently and updated automatically every so often in the client GUI, rather than requiring various commands to be executed periodically.

Multiple client interfaces were created in order to demonstrate all of the provided features and abilities of the *DataGridService*. There is one main interface, which lists status information about the Data Grid. This interface contains forms for inputting new data as well as querying for records based on the DICOM meta-data.

The second interface is designed to make things more realistic. It mimics the interface of an actual MR device scanner. So the user only enters the modality, patient name, and body part, since the rest of the data is generated by the scanner.

For now, since the transfers are small and don't take too much time to execute, no threading is being used. When the user hits "submit" the application waits for a response from the *DataGrid* service. In the future, threading may be implemented since there may be data sets with multiple images being transferred, which will take considerably longer. For example, with skull readings, generally around 80 2-dimensional "slices" at different areas of the brain are taken per study.

4.1 Storage and Retrieval Interface

There are several ways of designing graphical user interfaces. One can use platform-specific code, toolkits that generate code that is compatible with different platforms, etc. Since all the source code for the *DataGridService* is in Java, and because the bulk of GT4 is written in Java, Java seemed like the logical decision for implementing the interfaces.

Even after deciding to use Java, another decision must be made; that is, which GUI-toolkit will be used? Sun Microsystems provides two with its Java releases, the *Abstract Widget Toolkit* (AWT) and *Swing*. *Swing* is a newer implementation which extends AWT with additional functionality and also provides a more appealing look and feel. The best thing about these two is that they are bundled with Java and thus code written for them will run on any machine that has a *Java Virtual Machine* (JVM). However, the overall look and feel of many of the "widgets" is different than the native look of the operating system or windowing environment, at least as of this writing.

There is another toolkit by IBM, the *Standard Widget Toolkit* (SWT), which addresses the above problem by making calls to native widgets. As a result, windows look exactly as they would if they were written for the native windowing environment. The only case in which this doesn't hold is if the native windowing environment does not have a widget that SWT supplies, in which case SWT provides its own. The drawback to SWT is that it is separate from the stock JVM. Therefore, to use it, the client application would need to bundle the SWT libraries, or make sure that clients have them installed on their machine.

Since user-friendliness is one of the main goals of this work, SWT has an advantage since it provides users with an interface they are comfortable with, no matter what platform

they are on. It is therefore the chosen toolkit for this interface. Illustration 11 shows a screenshot of the final look, running on Linux.

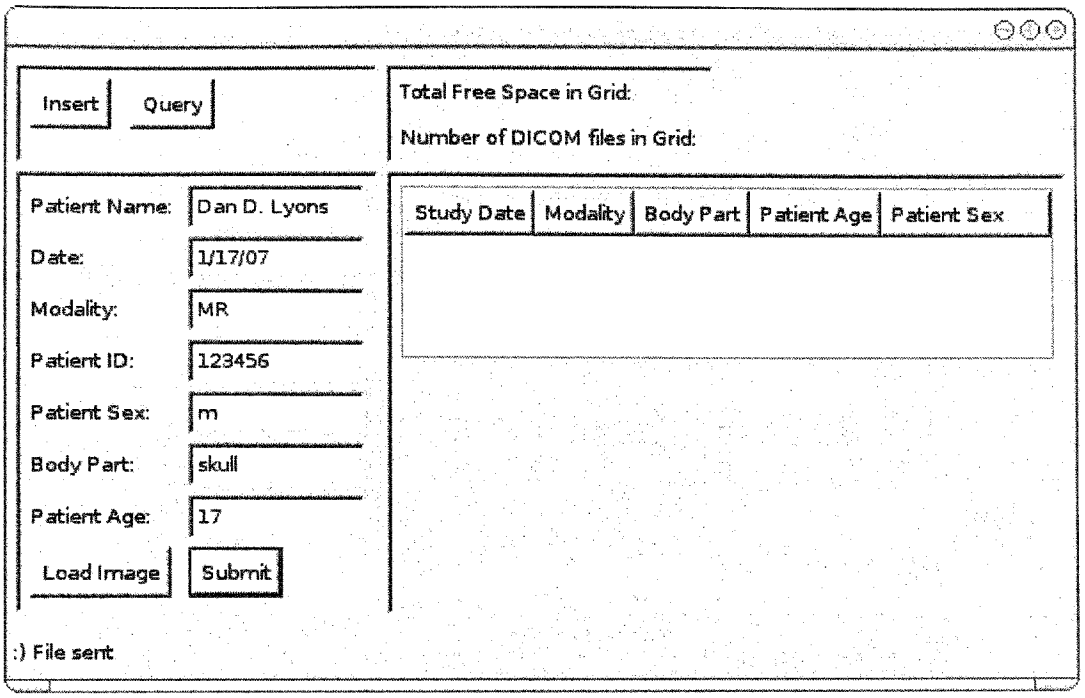

Illustration 11: The client interface

4.2 MRI Device Interface

For the MRI simulation device, an application was found, called "Virtual MRI" [36] that implements a virtual MRI device similar to an actual scanner. This software was intended for medical students, who don't have access to MRI devices to learn, to use this environment to get familiarized with the machines. But it is also a perfect solution for demonstrating how such devices can be integrated with this *DataGrid* and *DicomWriter* implementation.

In order to achieve a grid-enabled virtual-MRI machine, some modifications had to be done to the build files and of course to some source code files of *Virtual MRI* (VMRI). The main class to be modified is virtual.mrt.VMRTFrame, which is the main part of the VMRI interface. Essentially what was done is override the default action associated with the "save" button. Now, when a user presses the "save" button, the DICOM file created by VMRI is sent to the Data Grid. This consisted of overwriting the action associated with pressing the button. Instead of opening up a standard file dialog, a new dialog was created where the user can enter patient information to associate with the image.

In order to make the DICOM object created by VMRI compatible with the *DataGridService*, some additional work had to be done. This is because VMRI uses a different DICOM toolkit, not *Pixelmed*. As a result, the values for the DICOM attribute needed to be extracted from the DICOM object created by dicomie, which is the toolkit used by VMRI. These extracted values then had to be set as attributes of a *DicomType* object, which can then be sent to the *DatabaseService*. The rest of the process was the same as for the command line clients: obtain the endpoint reference to the *DatabaseService* and call its *storeDicom* method with the created *DicomType* as the parameter.

4.3 Common Problems

One problem that occurred with both GUI clients was the fact that the clients weren't trusting the Certificate Authority of the host they were trying to connect to (i.e the cluster). This resulted in a *javax.net.ssl.SSLHandshakeException* in both of the interfaces. To address this problem, Mind's Certificate Authority key was added as a

trusted Certificate Authority in the system's *Java Development Kit* (JDK) using the Java-supplied *keytool* utility. This procedure must be replicated on all systems that need to run any graphical interface that connects to the Grid. An alternative would be to use a custom HTTPS handler in the client, but this would be considerably more complex to program and would require external Java libraries, such as the Apache Commons *HttpClient* to be installed on the client systems.

5 Conclusion

This thesis work has shown that Grid computing enables a superior platform for collaboration. It has also shown that Grid computing provides more efficient development of collaboration software, thanks to available open technologies that provide a lot of built-in functionality. The use of high-resolution video visualization and collaboration was also made possible by building and setting up a tiled-display wall.

Overall, the work has shown a way to share a high-performance computing facility for both compute power and date storage, in order to help solve large-scale scientific problems. In this case, the storage and computing facilities were one and the same, which improves performance since the data to be processed is already local on the machine, although in practice they can be totally separate entities.

The implemented application in seizure detection demonstrated how the components of the platform can be combined to solve a computationally and data intensive problem. The design of the system also showed how the system can be extended to solve different types of problems as well.

LIST OF REFERENCES

[1] I. Foster, C. Kesselman, S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations", *International J. Supercomputer Applications*. Vol. 15. pp. 200-222, 2001.

[2] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration". The Globus Alliance. 2002.

[3] D. Twiddy, "Medical Records Going Electronic". *South Florida Sun-Sentinel.* November 2006.

[4] K.K. Jerger, T.I. Netoff, J.T. Francis, T. Sauer, L. Pecora, S.L. Weinstein, and S.J. Schiff, "Early Seizure Detection", *Journal of Clinical Neurophysiology*. Vol. 18. pp. 259-268, 2001.

[5] B. Litt and J. Echauz, "Prediction of epileptic seizures", *The Lancet Neurology*. Vol. 1. pp. 22-30, 2002.

[6] M.T. Tito, M. Ayala, I. Yaylali, M. Cabrerizo, A. Barreto, N. Rishe, M. Adjouadi, "Can EEG Processing Reveal Seizure Prediction Patterns?", IASTED Graphics and Visualization in Engineering, pp. 47-52, Clearwater, FL, USA. Jan. 2007.

[7] "BBFTP File Transfer Protocol". *http://doc.in2p3.fr/bbftp.*

[8] J.F. Lingling, C. Wasson, and G. Humphrey, Dept. of Comput. Sci., Virginia Univ., Charlottesville, VA, USA, "Toward seamless grid data access: design and implementation of GridFTP on .NET", *Grid Computing*. Vol. 6. pp. 164-171, 2005.

[9] R. McClatchey, D. Manset, T. Hauer, F. Estrella, P. Saiz, D. Rogulin. "The *MammoGrid* Project Grids Architecture", Conference for Computing in High-Energy and Nuclear Physics (CHEP 03), La Jolla, California. March, 2003.

[10] A. Rajasekar, M. Wan, R. Moore, W. Schroeder, G. Kremenek, A. Jagatheesan, C. Cowart, B. Zhu, S. Chen, and R. Olschanowsky, "Storage Resource Broker - Managing Distributed Data in a Grid", *Computer Society of India Journal*. Vol. 33. pp. 42-54, 2003.

[11] H. Duque, J. Montagnat, J.M. Pierson, L. Brunie, and I.E. Magnin, "DM2: A Distributed Medical Data Manager for Grids", *ccgrid*. Vol. 00. pp. 606, 2003.

[12] J. Jovicich, M.F. Beg, S. Pieper, C.E. Priebe, M.I. Miller, R.L. Buckner, and B. Rosen. "Biomedical Informatics Research Network: Integrating Multi-Site Neuroimaging Data Acquisition, Data Sharing and Brain Morphometric Processing", CBMS, pp. 288-293, 2005.

[13] P. Peters, J. Andreas, P. Saiz, and P. Buncic, "*AliEn*FS - a Linux File System for the *AliEn* Grid Services", *ECONF*. Vol. C0303241. pp. THAT005, 2003.

[14] "File System in Userspace". *http://fuse.sourceforge.net.*

[15] I. Foster, "What is the Grid? - a three point checklist". *http://www.gridtoday.com/02/0722/100136.html.*

[16] W. Blanke, C. Bajaj, D. Fussel, and X. Xhang, "The Metabuffer: A scalable Multiresolution Multidisplay 3D Graphics System using Commodity Rendering Engines". University of Texas at Austin. 2000.

[17] L. Renambot , B. Jeong, R. Jagodic, A. Johnson, and J. Leigh. "Collaborative Visualization Using High-Resolution Tiled Displays", CHI 06 Workshop on Information Visualization and Interaction Techniques for Collaboration Across Multiple Displays, Montreal, Canada. April, 2006.

[18] Y. Lai, I. Osorio, M.F. Harrison, and M.G. Frei, "Correlation-dimension and autocorrelation fluctuations in epileptic seizure dynamics", *The American Physical Society*. Vol. 65. pp. 65-69, 2002.

[19] M.C. Casdagli, L.D. Iasemidis, R.S. Savit, R.L. Gilmore, S.Roper and J.C. Sackellares, "Non-linearity in invasive EEG recordings from patients with temporal lobe epilepsy", *Electroencenphalogr. Clin. Neurophysiol.* Vol. 102. pp. 98, 1997.

[20] "LA-Grid". *http://latinamericangrid.org.*

[21] J. Delgado, M.R. Guillen, M. Lahlou, M. Adjouadi, A. Barreto, and N. Rishe. "MIND: A Tiled Display Visualization System at CATE FIU", IASTED Conference on Visualization in Engineering, pp. 68-73, Clearwater, FL, USA. January, 2007.

[22] G. Humphreys, M. Houston, Y. Ng, R. Frank, S. Ahern, P. Kirchner, and J. Klosowski. "*Chromium*: A Stream Processing Framework for Interactive Graphics on Clusters", Siggraph, pp. 693-702, 2002.

[23] B. Jeong, L. Renambot, R. Singh, A. Johnson, J. Leigh, "High-Performance Scalable Graphics Architecture for High-Resolution Displays". Electronic Visualization Laboratory, University of Illinois at Chicago. 2005.

[24] "Programmer's Manual for Shared Applications". Access Grid Tool-kit Documentation. *http://www-unix.mcs.anl.gov/fl/research/accessgrid/documentati on/SHARED_APPLICATIONS_MANUAL/ProgrammersManual_SharedApplicati .. onsHTML .htm.* January 2004.

[25] "Virtual Venues Architecture 2.0". The Futures Laboratory, Argonne National Laboratory, http://www-new.mcs.anl.gov/fl/re search/accessgrid/documentation /VirtualVenuesArchitectureDRA FT.doc.

[26] "NPACI Rocks Viz Roll". http://www.rocksclusters.org/roll-documentation/viz/ 4.2.1.

[27] "GNU Compiler Collection (GCC) C-compiler documentation", *http://gcc. gnu.org/onlinedocs/.*

[28] M. Wilding and D. Behman. Self-Service Linux: Mastering the Art of Problem Determination, ISBN: 013147751X. Prentice Hall Ptr. Upper Saddle River, NJ. 2005.

[29] "GT 4.0 Reliable File Transfer (RFT) Service". *http://www.globus.org/toolkit/ docs/4.0/data/rft/index.pdf.* The Globus Alliance. 2006

[30] J. Novotny, S. Tuecke, and V. Welch, "An Online Credential Repository for the Grid: *MyProxy*", *Proceedings of the Tenth International Symposium on High Performance Distributed Computing (HPDC-10).* Vol. 10. pp. 104-111, 2001.

[31] B. Sotomayor and L. Childers. Globus Toolkit 4: Programming Java Services, ISBN: 0123694043. San Fransisco, CA. 2006.

[32] "Digital Imaging and Communications in Medicine Standard(DICOM)". *http://medical.nema.org/dicom/2006.* 2006.

[33] D. A. Clunie, "*Pixelmed* DICOM API for Java". *http://www.dclunie.com /Pixelmed/software/*

[34] Apache Software Foundation, "Apache Axis *MessageElement* Documentation". *http://ws.apache.org/axis/java/apiDocs/org/apache/axis/messge/MessageElement. html.*

[35] M. Antonioletti, M.P. Atkinson, R. Baxter, A. Borley, N.P. Chue Hong, B. Collins, N. Hardman, A. Hume, A. Knox, M. Jackson, A. Krause, S. Laws, J. Magowan, N.W. Paton, D. Pearson, T. Sugden, P. Watson, and M. Westhead, "The Design and Implementation of Grid Database Services in OGSA-DAI", *Concurrency and Computation: Practice and Experience.* Vol. 17. pp. 357-376, February 2005.

[36] T. Hacklaender, C. Schalla, A. Truemper, H. Mertens, J. Hiltner, and B.M. Cramer, "A Virtual MR Scanner for Education". Proceedings of RSNA 2002, 88th Scientific Assembly and Annual Meeting, Chicago, USA. 2002

[25] "Virtual Venues Architecture 2.0". The Futures Laboratory, Argonne National Laboratory, http://www-new.mcs.anl.gov/fl/re search/accessgrid/documentation /VirtualVenuesArchitectureDRA FT.doc.

[26] "NPACI Rocks Viz Roll". http://www.rocksclusters.org/roll-documentation/viz/ 4.2.1.

[27] "GNU Compiler Collection (GCC) C-compiler documentation", *http://gcc. gnu.org/onlinedocs/.*

[28] M. Wilding and D. Behman. Self-Service Linux: Mastering the Art of Problem Determination, ISBN: 013147751X. Prentice Hall Ptr. Upper Saddle River, NJ. 2005.

[29] "GT 4.0 Reliable File Transfer (RFT) Service". *http://www.globus.org/toolkit/ docs/4.0/data/rft/index.pdf.* The Globus Alliance. 2006

[30] J. Novotny, S. Tuecke, and V. Welch, "An Online Credential Repository for the Grid: *MyProxy*", *Proceedings of the Tenth International Symposium on High Performance Distributed Computing (HPDC-10).* Vol. 10. pp. 104-111, 2001.

[31] B. Sotomayor and L. Childers. Globus Toolkit 4: Programming Java Services, ISBN: 0123694043. San Fransisco, CA. 2006.

[32] "Digital Imaging and Communications in Medicine Standard(DICOM)". *http://medical.nema.org/dicom/2006.* 2006.

[33] D. A. Clunie, "*Pixelmed* DICOM API for Java". *http://www.dclunie.com /Pixelmed/software/*

[34] Apache Software Foundation, "Apache Axis *MessageElement* Documentation". *http://ws.apache.org/axis/java/apiDocs/org/apache/axis/messge/MessageElement. html.*

[35] M. Antonioletti, M.P. Atkinson, R. Baxter, A. Borley, N.P. Chue Hong, B. Collins, N. Hardman, A. Hume, A. Knox, M. Jackson, A. Krause, S. Laws, J. Magowan, N.W. Paton, D. Pearson, T. Sugden, P. Watson, and M. Westhead, "The Design and Implementation of Grid Database Services in OGSA-DAI", *Concurrency and Computation: Practice and Experience.* Vol. 17. pp. 357-376, February 2005.

[36] T. Hacklaender, C. Schalla, A. Truemper, H. Mertens, J. Hiltner, and B.M. Cramer, "A Virtual MR Scanner for Education". Proceedings of RSNA 2002, 88th Scientific Assembly and Annual Meeting, Chicago, USA. 2002