A Unified Data Grid Replication Framework

Tim Ho and David Abramson
{tim.ho, davida}@csse.monash.edu.au

Monash e-Science and Grid Engineering Lab
Faculty of Information Technology
Monash University,
900 Dandenong Rd, Caulfield East, 3145, Australia

Abstract

Modern scientific experiments can generate large amounts of data, which may be replicated and distributed across multiple resources to improve application performance and fault tolerance. Whilst a number of different replica management systems exist, particular communities usually adopt a single system. This creates problems when an application program spans more than one community, because it may need to target more than one middleware layer. One solution to this problem is to build a more flexible data access layer above the specific replica middleware. In this paper, we discuss such an architecture, the Grid Replication Framework, which provides applications with an abstract interface to existing replica systems. Further, the framework's flexible plug-in architecture makes it easy to support new middleware as it becomes available.

1. Introduction

Grid middleware enables the sharing of geographically distributed resources, such as high performance computers, data archives and scientific instruments, and provides mechanisms that unify access by hiding much of the inherent heterogeneity [7]. Such middleware not only provides access to high-end computational data and instrumentation services, but also allows users to perform their tasks with little regard to geographic location.

Scientific applications, such as high energy physics [23][25], geophysics [21][26] and astronomy [24][27], often generate large amount of data, and users who require access to the data are sometimes geographically dispersed. Such data can be replicated and stored in different locations in order to optimize data access performance and provide better fault tolerance. Data replicas can significantly reduce the data access latency, and hence improve the program performance.

Data replication in the Grid has attracted a great deal of interest [1][2][3][10][12]. In general, a replica management system provides mechanisms to search, locate and access existing data, as well as to create and register new data. It also defines how multiple data replicas can be identified, and how the user may access such data. The latter may consist of different access level and/or security features (e.g. user certificates) in order to control who is allowed to access the data. There are a few research projects [15][18][19] that explore optimal methods for choosing which replica to access. These systems rely on various metrics, such as network conditions, distance between the application and the data, dataset size, average file transfer time, and overall performance of the remote servers.

Several Grid-based data replication systems exist, including Grid Datafarm [12], Globus RLS [11] and SRB [3], to name a few. These systems each have their advantages and disadvantages, but have been widely adopted by specific communities. One common shortcoming is that they use different protocols for accessing data, and thus, clients for one replica system usually cannot easily access data stored by another system. In this case, third party tools, such as GridFTP [17] distributed in the Globus Toolkit [28], are often required to move the data across the network.

This drawback can be significant. Imagine a scientific project that has distributed terabytes of experimental results using, say, the SRB. Later, the project is involved in a collaboration based on Gfarm. In order to share and analyse the data, the users of both projects have several choices. They may agree on using one replica system and move all data to it, or they can employ third party file copy tools, and/or modify the scientific applications to read data from both systems. Moving data may be expensive and can lead to incoherence problems when the data is updated. Moreover, a program may only require access to small portion of a large file, in which case copying the entire file can be inefficient. On the other hand, code modification tends to be error prone. This also means that users of one system must also have knowledge of the other. Clearly, a higher level middleware that provides an abstract interface to the different replica systems would offer significant flexibility for application developers.

We have developed a Grid Replication Framework (GRF) that unifies and provides a common API to a

number of current replica management systems. The GRF has a flexible architecture so that users can add new clients for other replica systems without modifying the framework itself. The API builds on existing IO primitives and provides applications transparent access to the replicated data stored in supported replica servers. When an application is developed on top of GRF, users specify which replicated data should be accessed by placing some meta data in a mapping file. Thus, without modifying the application code, the user can alter the path to the dataset and instruct GRF to read data from any supported replica servers. We have demonstrated the effectiveness of the GRF with our own GriddLeS Replication Service [5][16]. By leveraging GriddLeS over the GRF, applications can access a range of different replica management systems transparently without source code modification.

In addition to providing a single uniform API, the GRF also provides optimization mechanisms for improving the data access performance. It does this by monitoring the network bandwidth and latency to the various replica servers, using a tool like the Network Weather Service (NWS) [14], and it then dynamically chooses the most appropriate replica during execution.

In this paper, we discuss the components of the GRF, and its plug-in architecture. We also demonstrate that the framework has a flexible design to support new replica systems.

2. Access to replicated data

As mentioned, scientific data is often replicated and distributed. Managing replicated data is a non-trivial task Thus, several replica management systems have been developed to tackle issues such as user access control, scalability, fault tolerance and performance.

The Storage Resource Broker (SRB) [3] is a data management system that employs a client-server architecture. It provides mechanisms to access data based on some attributes other than filenames. SRB uses a MCAT (Meta data Catalog) service to store meta data information for the stored datasets. Such information is used to identify and describe the associated data. Users of SRB can access the data using the command line clients called Scommands. Other means including a web portal called MySRB, and a Windows-based client called inQ. There are also APIs in C, Java and Python. SRB also provides two libraries, SrbIO and UnixIO, to facilitate code modification for legacy programs.

Initiated in Japan, Gfarm is an implementation of the Grid Datafarm architecture [12]. This architecture is designed to handle hundreds of terabytes to petabytes of data using a global distributed file system. Different from SRB, Gfarm focuses on a Grid file system that provides scalable IO bandwidth and scalable parallel processing by integrating many local file systems and clusters. It uses a meta data management system to manage the file distribution, file system meta data and parallel process information. A file system daemon runs in each node to support remote file operations. The daemon also handles user authentication, file replication, node resource control and status monitoring. Files stored in the Gfarm file system can be accessed using a C library, or the Gfarm command line clients. Also, a syscall-hook library is provided for legacy programs to access Gfarm files.

The Globus Replica Location Service (RLS) [11] is an implementation of the RLS framework [1]. RLS is designed to replace the centralized Globus Replica Catalog in previous version of the Globus Toolkit. The RLS maintains and provides the mapping information from unique logical file names to physical file names; each of the physical file names usually contains the actual location of a file. It provides API and client commands to register, query and remove the file mappings but does not provide any data access mechanism. Other tools are needed in order to access the data. Files registered in the RLS can be copied using other tools such as GridFTP [17] and, as a technical preview, the Data Replication Service (DRS) [20]. Despite this, there is no support provided by RLS to perform simple file IO.

For most applications data access usually involves conventional file operations, such as open, read, write, seek, stat and close. Each of the above three systems are built using different standards and protocols. Therefore, data access to one system is different from the others.

SRB and Gfarm both provide APIs for accessing replicated data and the APIs are in Unix IO style. Thus, performing file IO on an SRB object or a Gfarm file is similar to performing local file IO. However, this can create problems in research collaboration when shared data are stored in different systems, because an application developed to read SRB data cannot read files stored in Gfarm. Clearly, a more flexible data access middleware on top of the systems is needed.

Such middleware should provide transparent data access to the application. This means that the application is not required to know the location of the file or how to access it. An example of such middleware that has similar goals is GFAL [22], which is a library that presents Unix IO style interface for IO operations to provide file access using logical file names rather than fixed file names. However, one significant disadvantage in GFAL is that it does not support multiple replica systems, including SRB and

Gfarm, which means that the application cannot access data stored in different replica systems.

GFAL has another limitation, namely, no optimisation is performed during file access. Optimised data access can significantly increase the execution performance, as presented in [16]. Optimisation methods varies, but the goal is to locate a better data source using information such as network condition, CPU load, distance to the data, etc.

3. The Grid Replication Framework

GRF unifies the existing replica systems and hides the underlying complexity of the different data access mechanisms. Our current implementation supports , SRB, Globus RLS and Gfarm. Building on top of GRF, a Grid application is provided with a set of Unix style IO functions that support transparent data access. The file IO operations are separated from the application itself, so that there is no need to modify the application code when change of location occurs.

Unlike the GriddLeS Replication Service, also implemented by our research group, the GRF has a flexible modular design for supporting new replica clients. With this plug-in design, users can easily develop clients for GRF that add data access supports for other systems.

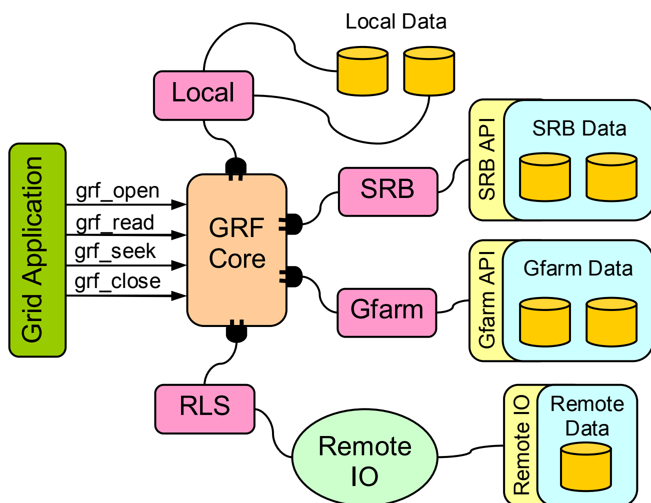

Figure 1 Architecture of GRF

3.1 Architecture of the framework

The GRF consists of a number of plug-ins that provide data access mechanisms for each replica system. As shown in Figure 1, a Grid application uses the GRF interface to perform desired IO operations. This interface abstracts the interfaces to the replica systems so that the application is not required to know about where the data is located, and how the data can be accessed.

In fact, the core of GRF does not implement any file access mechanism. Instead, the actual file operations are provided by replica client plug-ins. Currently, there are four plug-ins developed: local file access, SRB replica access, Globus RLS replica access, and Gfarm replica access. These plug-ins are used to provide system-dependent data access to the various replica systems. Without proper data access plug-ins, GRF will not perform any IO operations.

The framework consists of several components, including an IO interface, two configuration files, a Data Selection Service (DSS) and plug-ins that provide system-specific file access. Figure 2 shows the major

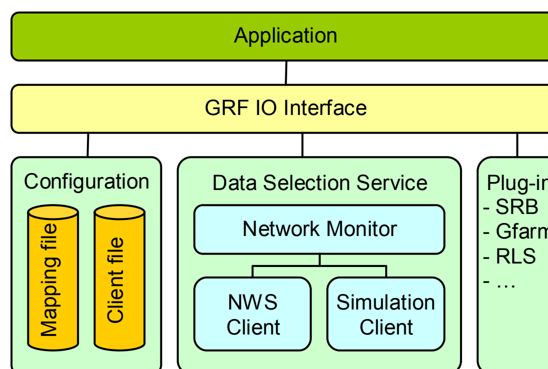

Figure 2 Components of GRF

components in the framework.

The GRF IO interface consists of a number of routines for performing IO on supported systems. The current implementation supports *grf_open*, *grf_read*, *grf_write*, *grf_stat*, *grf_seek* and *grf_close* operations.

The mapping file contains the mappings from keys to the physical paths of the files. A key is a unique identifier that serves as a logical file name used by the Grid application. During the *gfs_open* call, the GRF uses the key given by the application to find the data location from the mapping file. Below is an example (on one line) that instructs GRF to access an SRB object, where *input* is the key.

```
myhost.monash.edu.au:input
srb:srbhost.monash.edu.au:5544:/mona
sh-srb/home/tim.srb:input.dat
```

Each entry in the mapping file consists of two main parts separated by an empty space. The first part is an identifier that matches the key used in the *grf_open* call. The second part consists of a client key used by GRF to identify which plug-in is used for the IO;

information after the first colon is system-specific and is parsed and used by the plug-in. The client key refers to the library path information stored in the client file. The GRF examines the client file and finds out the full path to the plug-in shared library. The library is loaded during runtime for any supported IO operations and the system-specific file details are passed to the corresponding routines. An example of an entry in the client file is as follows:

```
srb=/home/tim/grf/io/libsrb.so
```

The mapping file and the client file are two key components because they contain the mapping to the actual data location as well as the client library location. When change of file location occurs, the user only modifies the mapping file, rather than changing the application code. When a new plug-in is added, only the client file requires changes. Avoiding code modification is a major advance to the conventional IO routines and provides data access flexibility to the application running on the Grid.

The GRF employs a Data Selection Service (DSS) for the read operations. This service uses a network monitor to continuously monitor the network condition to the data servers and dynamically switch to a better one based on network statistics. This network monitor includes a NWS client that retrieves the NWS measurements and generates network condition forecasts. When a better server is found based on the NWS predictions, GRF will dynamically switch to, and source the data from, the better one. Because this selection is made dynamically during runtime, the program execution time can be significantly shortened. DSS is an optional feature and can be disabled when no optimisation is required.

Since the file offset information for replicas are stored independently in each replica server, change of replica source may result in accessing incorrect blocks of data. To avoid this, the DSS employs proper synchronization for file access every time when a replica selection is done. The file offset is stored and updated in GRF at the end of each IO operation. If there is a change to the server, then GRF will seek to the correct offset before accessing the replica.

For evaluation purposes, simulated network statistics can also be used. Apart from real time NWS forecasts, the network monitor can also use a user-defined network statistics for simulation and evaluation purposes. Currently, network bandwidth is used for deciding which replica server is used. When needed, users can define a sequence of network bandwidth in a file and the DSS will select a server based on the information provided in that file.

3.2 Data access mechanisms

The current GRF implementation supports four different access mechanisms: local file access, SRB access, Gfarm access and RLS access. Each of them provides specific file IO for the corresponding system.

3.2.1 Local file access

There are two cases where the application using GRF may access a local file. The first is when the user wants to use conventional IO for local files. In this case, a full path beginning with a forward slash ("/") to a local file should be specified in *grf_open*.

The other case is when a key is specified but the mapping actually refers to the local file plug-in. The reason of doing so varies. One reason is that the user wants to take advantage of the flexibility of GRF because change of file location may occur. This can happen when the application requires access to a replica that is copied to the local machine for better performance. It also benefits if the application requires frequent change of file location.

In either case, conventional IO routines will be used. This is a straightforward task because the arguments and return values of the GRF IO are similar to the conventional one. Yet, there are two major differences.

The first difference is the file descriptor (FD) used in the two cases. For the first one, the FD is exactly what the operating system returns. Basically, when a file path beginning with a slash is specified, the GRF simply passes the arguments to the corresponding Unix IO routines and returns what the routine returns.

However, for the second situation, a GRF-specific FD will be used. The main reason is that in subsequent IO operations GRF requires this FD to find out whether the operations are GRF IO or Unix IO. In addition, this GRF-specific FD is a simple integer that emulates the FD normally returned by the *open* function.

Another difference is that when a GRF client library is used, a GRF-specific replica object is created. The object is used to store information including the client module, the key for the mapping, the attached network monitor, the file current offset, and most importantly a handle to a replica system-specific object that contains information such as the server location, port, path, etc. The replica object is identified later using the FD.

To use the local file access plug-in, a local file entry should be added to the mapping file. This looks like the following:

```
myhost:input local:/tmp/input.dat
```

In this example, the host address (where the program running in) and the key identifier are on the left hand

side, where *input* is the key that would be specified in the *grf_open* call. On the right hand side, *local* is the key to identify the client library. The other information on the entry is the full path to the local file. All file operations using *input* as the key will be performed on this file.

3.2.2 SRB replica access

A SRB plug-in, which has similar functions to the one used previously in GriddLeS [16], is also implemented. When an application requires access to data stored in the SRB, the current implementation of the SRB client plug-in uses the standard SRB user authentication (i.e. requires .MdasEnv and .MdasAuth) to connect to the specified SRB server. When the server is connected, the client will obtain a list of available replicas from the SRB server. The GRF then creates an object that represents the list of replicas and proceeds to open the SRB files. When required, a network monitor is started and attached to the GRF object. At the end of the open function, *grf_open* returns a unique GRF-specific file descriptor (FD) that emulates the FD normally returned by the conventional *open* function.

Below is an example of an entry in the mapping file that uses the SRB plug-in. The key identifier on the left hand side is omitted.

```
srb:mySrbhost:5544:/srb/home/tim.srb:
input
```

On this entry, the word *srb* refers to the SRB client library and other required information includes the SRB server address, port, path to object, and the object identifier. Since this information is specific to SRB, it is the job of the SRB plug-in to parse this information. GRF does not have knowledge about the details and it merely passes this to the SRB plug-in.

3.2.3 Gfarm replica access

A Gfarm plug-in provides a GRF interface for the Gfarm file system. Similar to SRB, Gfarm provides well defined C interfaces that have IO functions similar to Unix-style IOs for performing remote file operations. Since the Gfarm execution environment must be initialized before any file operation, the first task in the open call is to initialize the Gfarm environment. After that, the file path (i.e. Gfarm URL) is used with the Gfarm open routine, *gfs_pio_open*. The returned Gfarm file handle is stored in the GRF replica object and this replica object will be stored within GRF for subsequent IO operations. Usual Gfarm user authentication is used; the configuration file .gfarmrc is required.

The Gfarm plug-in does not use GRF's replica selection feature because when a Gfarm file is accessed, the Gfarm file system daemon automatically selects the best replica based on average CPU load and response time. Thus, optimization is done by Gfarm.

The following is an example of an entry in the mapping file for accessing a Gfarm file:

```
myhost:input gfarm:input.dat
```

In this case, when the application is opening a file using the key *input*, GRF passes the file information to the Gfarm plug-in and perform the IO operations on the Gfarm file.

3.2.4 Globus RLS replica access

The Globus RLS plug-in for GRF is considerably different from the other two discussed to date, because RLS lacks a data access API. Although files registered in RLS can be copied using other tools, a set of Unix-like IO routines would be preferred when an application requires access to a file (probably a small portion), and to perform IO operations, such as stat and seek. This is especially true for a workflow that does file-based (and potentially replica-based) communication on the Grid, when no file copy is desired.

As a result, a simple remote IO interface that provides file IO access to the remote data is implemented in the RLS plug-in. This interface is a web service that exposes simple IO functions on remote files. The plug-in performs these operations via gsoap [13]. One reason of using a web service is because of firewall issue. Direct socket connection may be more straightforward but the port for the connection is often blocked by firewalls. Each machine that contains the file must be running the web service in order to provide remote IO access. Certainly, scalability can be a problem when many replicas exist. This interface is considered as experimental since it does not address common problems on IO over wide area networks, including access control, scalability and remote IO performance. A reliable IO library that provides remote data access should be employed in a production environment. This library should, ideally, address issues such as security, portability, and performance, as in [6] and [8],

Similar to Gfarm, the Globus RLS module requires activation. When a file is being opened, the RLS plug-in firstly ensures that the module is activated, and then connects to the RLS server to get a list of physical file paths using the given logical file name, which is given in the mapping file. The list is stored in the GRF replica object. A physical file path should contain the

web service address, port, and the path of the file. The plug-in communicates with the web service and makes sure the file exists and is accessible. At the end, when requested, a network monitor is run and attached to the GRF replica object.

Below is an example of an entry in the mapping file to access replica registered in RLS:

```
myhost:input rls:rlshost:39281:data
```

Apart from the RLS server address and port, the major information is the logical file name, *data*. The plug-in uses it to retrieve a list of registered replicas from the RLS server. An example of the physical file path registered in RLS is as follows:

```
http://myhost:8080/RemoteIO:/home/tim
/input.dat
```

This tells the RLS plug-in that it needs to use the web service available on *myhost* to access the file *input.dat*.

4. Case Studies

Two case studies were conducted to illustrate the functionality and performance of GRF. For evaluation purpose, SRB, Gfarm, RLS and the Remote IO web service are all running locally. Each experiment involves running the program twenty times to get an average time. The standard deviation is also calculated.

4.1 A simple copy program

The first case study concerned a simple file copy program, built on top of GRF. This application reads data (8KB each time) from a file (140MB) and writes the data to a local file. The input file was copied to Gfarm and SRB, and was also registered in RLS. The locations of the files were stored in the mapping file. Several experiments were done for this case study and each experiment was run twenty times. During this case study, no source code modification was needed for the copy program because the file location and plug-in information are separated from the program. Only the mapping file required changes.

Plug-in	Average Time (sec)	Std. Dev.	IO Used by GRF
n/a	9.2	1.2	Unix IO without GRF
Local	9.5	1.3	Unix IO
SRB	11.9	1.1	SRB API
Gfarm	12.7	0.7	Gfarm API
RLS	15.7	1.3	Remote IO web service

Table 1 Experiment results of the file copy program

The experiment results are shown in Table 1. In the first experiment, the program was run without GRF, and conventional Unix IO was used. For the plug-in experiments, the program was run and different plug-ins were used. Not surprisingly the GRF local access plug-in has the fastest execution time comparing to the other three due to the minimal overheads within the IO. In fact, the average execution time was very similar to the conventional IO. Note that the data was stored in a local server, and thus there is no network overheads involved. Rather, the result shows that overheads in the IO layers for the replica systems would result in significantly decreased performance.

In addition, Linux commands *cp* and *scp*, as well as SRB's *Sget* command, were used to copy the same 140MB file locally in order to further compare the performance. The results are shown in Table 2. As expected, the overall performance of the plug-ins was slower than *cp* but clearly faster than using *scp*. The SRB client, *Sget*, uses multiple streams to retrieve data from the SRB server and thus it should run faster than the SRB plug-in. Yet the results show that it is not. The reason for this is unknown.

Commands	Average Time (sec)	Std. Dev.	System
cp	9.8	1	Linux
scp	19.3	0.9	Linux
Sget	14.4	1.2	SRB

Table 2 Copy 140MB using different commands

4.2 CCAM - a global climate model

The second case study concerned a global climate model, called CCAM [9]. CCAM is written in Fortran and compiled by the Intel Fortran Compiler. This model analyses the atmospheric effects over a particular area. It requires several input files during execution and writes output data to a file. Two experiments were conducted. The first experiment was to run CCAM normally without GRF for twenty times. During the second experiment, CCAM was run twenty times using GriddLeS. No code change was done in CCAM because GriddLeS allows running legacy programs on the Grid by intercepting and replacing the IO calls with the corresponding GRF IO routines.

As shown in Figure 3, some input files (such as topography and ground vegetation information) were stored in Gfarm, while the output data was written to an SRB server. Other input files exist locally, so GriddLeS and GRF were not needed. This scenario is similar to multiple organizations running applications that require shared data stored in different replica systems. With the use of GRF, the locations of the files

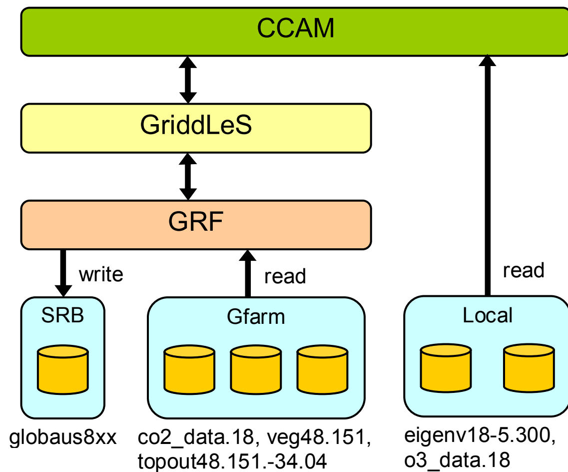

Figure 3 CCAM experiment

do not affect how the application is performed and the users are provided a flexible way to alter the file locations. Results are shown in Table 3. Average execution time increased from ~20 minutes to ~28 minutes. The execution time increased because CCAM was accessing data from different data source and various overheads exist in the IO layers.

Exp.	Average Time (min)	Standard Deviation
1	19.8	0.9
2	28.1	2.3

Table 3 Experiment results for CCAM

5. Conclusion

This paper discussed the Grid Replication Framework (GRF), which unifies the replica systems by providing an interface that abstracts the different data access mechanisms. The same set of IO routines is used to access files available in different systems. This provides applications transparent data access to the supported systems and enables easy change of data source without the need to modify the application. Importantly, building such application requires little replica system knowledge because GRF hides the complexity of the underlying IO operations. The case studies demonstrated that GRF can greatly help applications running in multiple communities where files are distributed and stored in different replica systems.

Performance improvement is performed by a data selection service. Experiments in previous studies show that optimized data access can significantly increase execution performance. Results shown in this paper further show that performing large amount of reads on small blocks of data is inefficient on the Grid because of the extra overheads. In particular, overheads within the IO layers of the different replica systems vary and can result in significant performance impact. Techniques such as pre-reading the data and buffering are needed. We plan to develop a module for GRF that provides a local data cache in order to overcome the overhead issue in the IO layers.

Reference

[1] A. Chervenak, E. Deelman, I. Foster, L. Guy, W. Hoschek, A. Iamnitchi, C. Kesselman, P. Kunszt, M. Ripeanu, B. Schwartzkopf, H. Stockinger, K. Stockinger, B. Tierney. *Giggle: A Framework for Constructing Scalable Replica Location Services*, SC2002, Baltimore, 2002.

[2] A. Samar and H. Stockinger, *Grid Data Management Pilot (GDMP): A Tool for Wide Area Replication*, IASTED International Conference on Applied Informatics (AI2001), Innsbruck, Austria, February 2001.

[3] C. Baru, R. Moore, A. Rajasekar, M. Wan. *The SDSC Storage Resource Broker*, Proceedings of CASCON'98 Conference, Toronto, Canada, 1998.

[4] D. Abramson, J. Giddy and L. Kotler, "High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?", International Parallel and Distributed Processing Symposium (IPDPS), pp 520- 528, Cancun, Mexico, May 2000.

[5] D. Abramson and J. Komineni. *Interprocess Communication in GriddLeS: Grid Enabling Legacy Software*, Technical report, School of Computer Science and Software Engineering, Monash University.

[6] I. Foster, D. Kohr, Jr., R. Krishnaiyer and J. Mogill, *Remote I/O: Fast Access to Distant Storage*, Proceedings of the Fifth Annual Workshop on I/O in Parallel and Distributed Systems, 1997.

[7] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers, USA, 1999.

[8] J. B. Weissman, M. Marina and M. Gingras, *Optimizing Remote File Access for Parallel and Distributed Network Applications*, Journal of Parallel and Distributed Computing, pp 1591-1608, 2001.

[9] J. L. McGregor, K. C. Nguyen and J. J. Katzfey, "Regional Climate Simulations using a Stretched-grid Global Model", Research Activities in Atmospheric and Oceanic Modeling, H. Ritchie (ed.), pp. 3.15-3.16, 2002.

[10] L. Guy, P. Kunszt, E. Laure, H. Stockinger, and K. Stockinger. *Replica Management in Data Grids*, Technical report, GGF5 Working Draft, July 2002.

[11] M. Manohar, A. Chervenak, B. Clifford, C. Kesselman, *A Replica Location Grid Service Implementation*, Data Area Workshop, Global Grid Forum 10, 2004.

[12] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, H. Sato, Y. Tanaka, S. Sekiguchi, Y. Watase, M. Imori, T. Kobayashi, *Worldwide Grid Data Farm for Petascale Data Intensive Computing*, Technical Report, Electro-technical Laboratory, ETL-TR2001-4, 2001.

[13] R. van Engelen and K. Gallivan, *The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks*, Proceedings of IEEE CCGrid Conference 2002.
[14] R. Wolski, N. Spring, and J. Hayes. *The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing*, Journal of Future Generation Computing Systems, Vol.15, No.5-6, pp.757-768, October, 1999.
[15] S. Vazhkudai, S. Tuecke, and I. Foster. *Replica Selection in the Globus Data Grid,* International Workshop on Data Models and Databases on Clusters and the Grid (DataGrid 2001), IEEE Computer Society Press, 2001.
[16] T. Ho and D. Abramson. *A GriddLeS Data Replication Service*, Proceedings of 1st International Conference on e-Science and Grid Computing, 2005.
[17] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke, *Data Management and Transfer in High-Performance Computational Grid Environments*, Parallel Computing, 2001.
[18] W. H. Bell, D. G. Cameron, L. Capozza, A. P. Millar, K. Stockinger, and F. Zini. *Design of a Replica Optimisation Framework*, Technical Report, DataGrid-02-TED- 021215, CERN, Geneva, Switzerland, 2002.
[19] Y. Zhao, and Y. Hu. *GRESS - a Grid Replica Selection Service*, ISCA 16th International Conference on Parallel and Distributed Computing Systems, PDCS-2003.
[20] Data Replication Service, http://www.globus.org/toolkit/docs/4.0/techpreview/datarep/
[21] EarthScope, http://www.earthscope.org/
[22] Grid File Access Library, http://www.gridpp.ac.uk/wiki/Grid_File_Access_Library
[23] GriPhyN, http://www.griphyn.org/
[24] NVO, http://www.us-vo.org/
[25] PPDG, http://www.ppdg.net/
[26] SCEC Community Modelling Environment, http://www.scec.org/cme/
[27] Sloan Digital Sky Survey, http://www.sdss.org/
[28] The Globus Toolkit, http://www.globus.org/toolkit/
[29] The Globus Toolkit Data Management, http://www.globus.org/toolkit/data/