A Virtual File System Interface for Computational Grids

Abdulrahman Azab and Hein Meling

Dept. of Electrical Engineering and Computer Science
University of Stavanger, N-4036 Stavanger, Norway

Abstract. Developing applications for distributed computation is becoming increasingly popular with the advent of grid computing. However, developing applications for the various grid middleware environments require attaining intimate knowledge of specific development approaches, languages and frameworks. This makes it challenging for scientists and domain specialists to take advantage of grid frameworks. In this paper, we propose a different approach for scientists to gain programmatic access to the grid of their choice. The principle idea is to provide an abstraction layer by means of a *virtual file system* through which the grid can be accessed using well-known and standardized system level operations available from virtually all programming languages and operating systems. By abstracting away low-level grid details, domain scientists can more easily gain access to high-performance computing resources without learning the specifics of the grid middleware being used. We have implemented such a virtual file system on HIMAN, a peer-to-peer grid middleware platform. Our initial experimental evaluation shows that the virtual file system only cause a negligible overhead during task execution.

1 Introduction

Grid computing is an appealing concept to support the execution of computationally intensive tasks; grid computing generally refers to the coordination of the collective processing power of scattered computing nodes to accommodate such large computations. Peer-to-peer grid computing extends this idea, enabling all participants to provide and/or consume computational power in a decentralized fashion. Although appealing at first sight, grid computing frameworks are complex pieces of software and thus typically come with some obstacles limiting adoption to expert programmers capable of and willing to learn to program using the grid middleware application programming interfaces (APIs). Therefore, a major challenge faced by developers of grid computing frameworks is to expand the base of grid application developers to include non-expert programmers, e.g. domain specialists using their own domain-specific tools. One approach is to devise transparency mechanisms to hide the complexities of the grid computing system.

In this paper, we propose the design of a programming interface for a peer-to-peer grid middleware through a *virtual file system* (VFS) [1,6,21]. The rationale

behind this idea is that the file system interface is well-known, and is accessible from just about any domain specific tool or even simple file-based commands. The approach also lends itself to devising a file system organization exposing varying degrees of complexity to its users. For example, complex management operations can be performed using one part of the file system hierarchy, whereas domain specialists can submit tasks through another part of the file system limiting the exposure to a most essential part of the system.

We have implemented such a VFS layer above a middleware core based on our HIMAN platform [11,13]. The VFS layer sits between the end user/programmer and the client program enabling them to assign a task to be executed on the grid simply by calling the `write()` function of the virtual file system on a specific virtual path. Similarly, when task execution has completed, the user can easily obtain the results by invoking the `read()` function on another virtual path. To evaluate the overhead imposed by the VFS layer, we compared a simple parallel matrix multiplication task running with and without the VFS layer. The results confirm our intuition that the VFS layer only adds a negligible computational overhead.

The rest of the paper is organized as follows: Section 2 gives an overview of the related efforts. Section 3 presents a general overview of grid middleware architectures. Section 4 describes the new added virtual file system layer and how it interacts with the other components in the middleware. Section 5 presents and discusses the results obtained from the performed experiments. Section 6 presents conclusions and future work plans.

2 Related Work

The concept of virtual file systems was first introduced in the Plan 9 operating systems [21] to support a common access style to all types of IO devices, be it disks or network interfaces. This idea has since been adopted and included in the Linux kernel [6], and support is also available on many other platforms, including Mac OS X [8] and Windows [1,4]. There are a wide range of applications of such a virtual file system interface, including sshfs, procfs, and YouTubeFS.

In the context of grid computing, providing access to a (distributed) file system architecture is both useful and commonplace. Transmitting the data necessary for computation is made easier with a distributed file system, and a lot of research effort has gone into data management for the grid, e.g. BAD-FS [14], and finding efficient protocols for data transfer, e.g. GridFTP [7]. BAD-FS [14] is different from many other distributed file systems in that it gives explicit control to the application for certain policies concerning caching, consistency and replication. A scheduler is provided with BAD-FS that takes these policies into account for different workloads. The Globus XIO [12] provides a file system-based API for heterogeneous data transfer protocols aimed for the Grid. This allows applications to take advantage of newer (potentially) more efficient data transfer protocols in the future, assuming they too provide the XIO APIs. These file systems tackle the problem of interacting with the data management component

and file transfer protocols of a data grid environment. However, none of them try to solve the generic interaction with the computational grid middleware for task submission, results collection in a suitable manner.

An interposition agent is a piece of software that inserts itself between two existing layers of software in order to modify their discourse [16]. In [17, 25], different roles of interposition agents are described and classified in more detail. Some interposition agents provide integration between two applications located on the same machine.

Other interposition agents provide seamless integration between two applications located on different machines. Parrot [25] provides seamless integration between standard Unix applications and remote storage systems. This integration makes a remote storage system appear as a file system to a Unix application. One drawback is that the user has to specify the location of the remote machine. Besides, all commands has to be included in a Parrot command, which decreases the provided transparency. Bypass [24] is a general purpose tool for building interposition agents based on split execution. Instead of locating an instance of the interposition agent on the home machine, the user interacts with a shadow process which communicates with the interposition agent on the remote machine.

Other examples are: GCB [23], DCache [15], Paradyn [20], and SOCKS [19]. All of these systems provide an intermediate layer between two applications to provide or enhance the integration or to perform an additional intermediate role. The user has to know how to deal with the front end application which is system specific. The proposed VFS layer provides an interface which is well known to all computer users and locates itself as the front end application. The user does not have to provide any information about the remote machine. In addition, it provides a data management interface (i.e. file system) for a computation management application (i.e. computational grid). To our knowledge, most of the existing computational grid middleware environments [2, 3, 5, 9, 10, 18, 19, 22] require a special language level API for interaction with the scheduler and to collect the results.

3 HIMAN Overview

The VFS layer has been implemented on HIMAN [13], a pure peer-to-peer grid middleware that support serial and parallel computational tasks. HIMAN has three main components: (i) a *worker* component responsible for task execution, (ii) a *client* component responsible for task submission and execution monitoring, and (iii) a *broker* responsible for the task allocation. All nodes in the grid have all three components. The client and broker of a submitting node is then responsible for task submission, allocation, and execution monitoring. Any node can submit tasks, and can also serve as executor for other tasks at the same time.

4 Virtualizing Grid Access

Existing grid computing platforms require that the user configure a wide range of parameters, have sufficient background in distributed computing, and proficient knowledge of the grid middleware APIs to be able to execute tasks on the grid. This might be appropriate for expert users, but for regular users, a simple and familiar interface is desirable. By far the most ubiquitous programming interface available on computer systems is the file system interface. Thus, our proposed VFS layer placed above our HIMAN grid computing middleware exposes such an interface to the application programmer, making it easy to interact with the middleware using a well-known interface. Moreover, it enables users to submit tasks using the `write()` function, and monitor and collect results from the computation using the `read()` function of the VFS layer.

4.1 The VFS Layer

Our VFS layer is implemented as an additional interface layer above the client using the Callback file system library [1]. Callback is similar to the FUSE library available on Linux and Mac OS X [6, 8], and enable building virtual file systems in user space. The Callback library contains a user mode API to communicate with user applications, and a kernel mode file system driver for communicating with the Windows kernel. The role of the Callback library in our VFS layer is to enable users to provide input files and collect result files using simple file system commands. In order to make the proposed VFS applicable and easy to implement in other grid systems which are based on different platforms, the communication between the VFS layer and the client is performed by means of simple UDP text-based commands. In order to implement VFS on another grid middleware, a small UDP communication module must be added to the client.

The installation of the VFS layer on a grid client machine is very simple. The user simply runs an MSI package, which installs the VFS layer as a Windows service. The service is configured to startup automatically when the user logs on, or manually by executing a `net start` command. Once the service is started, the VFS drive will be mounted. The VFS layer will communicate with the grid client upon the execution of specific file system commands. The virtual drive will be unmounted automatically when the user logs off, or it can be unmounted manually by executing a `net stop` command.

The VFS layer is composed of two main modules: Task Submission Module (TSM) and Result Collection Module (RCM).

Task Submission Module TSM contains two routines. The first is the virtual volume routine, and will be called when the Callback file system is mounted, and creates a virtual volume visible in Windows Explorer. This new volume will be used by the user to provide the input files. The second is the task submission routine and is responsible for forwarding the input files to the client in order to start the task submission process. Task submission requests are handled in the

`CbFsFsWrite()` function of the Callback user mode library, so that it is triggered only when a `write()` command for a specific file is executed on the virtual drive. The default case is when the program file is copied to the virtual drive. This can be changed by the user to determine which `write()` command will trigger the TSM. The task submission procedure depicted in Fig. 1 involves the following steps:

1. The user executes a `write()` command, through the file system interface (e.g. Windows Explorer), to write the input files into the virtual volume.
2. The file system interface forwards the command to the Windows kernel.
3. The kernel forwards the command to the Callback file system driver.
4. The file system driver responds by calling the `CbFsFsWrite()` function in the Callback user mode library.
5. This function invokes the task submission procedure in the TSM.
6. The task submission procedure responds by presenting the input files in the file list of the virtual volume, so that they can be accessed with the file system interface, and forwarding the input files to the client.
7. The client submits the task to the grid.
8. During the execution, the user can monitor the execution process (i.e. the progress) by executing a `read()` command on a specific text file on the virtual drive.

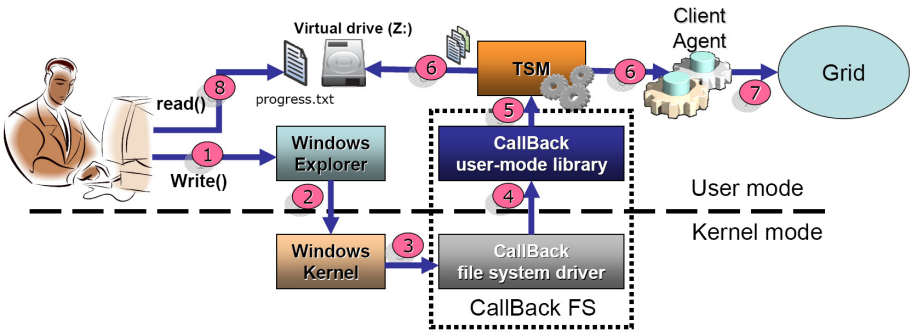


Fig. 1: Task submission procedure

Result Collection Module The objectives of the result collection module are twofold: (i) provide a file system interface through which the user can collect the result files from task execution, and (ii) signal the completion of task execution to the user. The result collection procedure depicted in Fig. 2 involves the following steps:

1. When task execution is completed, the results are sent to the client process on the submitting node [13].

2. The client sends the collected results to the RCM.
3. The RCM responds by creating a new virtual directory (e.g. `Results`) in the virtual drive, and creating virtual files in this directory referring to the result files. Creation of the `Results` directory indicates to the user that task execution has complete.
4. The user can collect the result files by executing a `move` command on the files in the `Results` directory to move the result files to a physical path.
5. Windows Explorer forwards the command to the kernel.
6. The kernel forwards the command to the Callback file system driver.
7. The file system driver responds by calling the `CbFsRenameOrMoveEvent()` event procedure in the Callback user mode library.
8. The `CbFsRenameOrMoveEvent()` procedure responds by moving the result files to the given physical path, and removing them from the file list of the `Results` directory of the virtual drive.

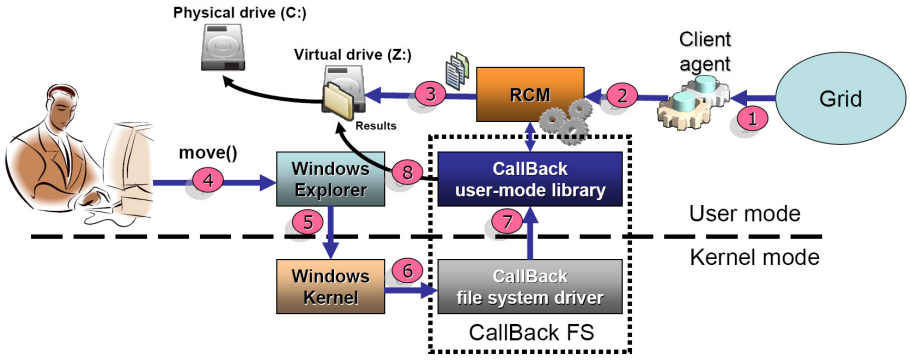


Fig. 2: Result collection procedure

4.2 A Simple Example

In this section we provide a simple example describing how to use the VFS layer for executing a parallel task of matrix multiplication. To accomplish this, the user must provide as input: an executable code file (e.g. `code.dll`), and the necessary data input files (e.g. `input?.data`). The number of input files correspond to the number of parallel subtasks that will be generated by the HIMAN middleware during execution of the task.

For illustration, the DOS command prompt is used (interactively) below to issue file system commands to interact with the grid middleware. The virtual drive is mounted on drive `Z:`, and the input files are stored in `C:\Input\`. The following steps describe the whole execution procedure, including results collection.

1. The user writes the data files to the virtual drive by typing:

   ```
   copy C:\Input\*.data Z:
   ```

2. The user writes the code file to the virtual drive by typing:

   ```
   copy C:\Input\code.dll Z:
   ```

   As explained in Section 4.1, this command will trigger the TSM to invoke the client to begin task execution, and to a create virtual file `progress.txt` on drive `Z:` for monitoring progress.

3. During the execution, the user can monitor the execution progress for the subtasks by typing:

   ```
   type progress.txt
   ```

   ```
   Subtask 1. Worker:193.227.50.201  Progress: 10%
   Subtask 2. Worker:74.225.70.20    Progress: 15%
   Subtask 3. Worker:87.27.40.100    Progress: 20%
   Subtask 4. Worker:80.50.96.119    Progress: 6%
   Subtask 5. Worker:211.54.88.200   Progress: 12%
   ```

4. The user can repeat the above command to keep up with the execution progress.
   The user need not consider management issues such as scheduling, fault tolerance, and connectivity. These issues are seamlessly handled by the HIMAN grid middleware [13]. In case of a worker failure, the worker address will be changed for the associated subtask in the progress file.
5. When the execution of one or more subtasks is completed, this will be revealed in the `progress.txt` file as follows:

   ```
   Subtask 1. COMPLETED
   Subtask 2. Worker:74.225.70.20  Progress: 90%
   Subtask 3. Worker:87.27.40.100  Progress: 88%
   Subtask 4. COMPLETED
   Subtask 5. COMPLETED
   ```

6. Upon the completion of all subtasks, the `Results` virtual directory will appear, enabling the user to move all the virtual files to a physical directory as follows:

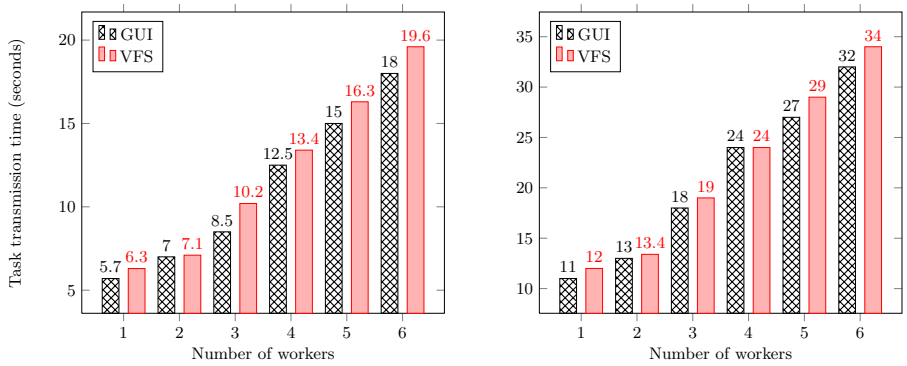
   ```
   move Z:\Results\*.* C:\Results
   ```

5 Initial Performance Evaluation

In order to demonstrate the applicability of our VFS layer, we have performed a simple experimental evaluation to reveal the overhead caused by the VFS layer

compared to interacting directly with the HIMAN middleware (through a GUI interface).

Two experiments were performed using a classic parallel matrix multiplication task for multiplying two square matrices of size: a) 1500×1500 and, b) 2100×2100. In both cases, different number of workers (i.e. parallel subtasks) varying from one to six were used. The results are shown in Fig. 3.



(a) Matrix size 1500×1500 (b) Matrix size 2100×2100

Fig. 3: Task transmission time against the number of workers using the client GUI and the VFS interface

In Fig. 3, it is clear that the computational overhead which is represented by the increase in the task transmission time in case of using the VFS interface instead of the built in client GUI, is nearly negligible. Since the transmission of the input files to workers is done in a serial fashion, the overhead is due to the time taken by the VFS to build the virtual files in memory and to communicate with the client for transmitting each of input files.

The two approaches, GUI and VFS, have identical CPU overhead simply because in both cases all pre-processing procedures are carried out by the grid client. The memory overhead is also identical since in both cases, the task input files are loaded into memory in order to be processed by the grid client upon the execution.

6 Conclusions and Future Work

Given the complexity of grid computing middleware, providing an easy to use interface for task submission is a significant challenge. In this paper, we have proposed a new technique for accessing computational grid middleware through a virtual file system interface. The proposed technique has been implemented on the HIMAN middleware, enabling non-expert users to submit compute tasks to

the grid. Our initial evaluation indicate that the overhead imposed by the VFS interface over the direct interaction approach is negligible.

Our current implementation support only serial task execution; in future work we will extend the VFS layer with support for parallel task execution. Moreover, we also plan to design a VFS-based grid portal that supports multiple grid computing frameworks, e.g. Globus and Condor. To accomplish this we will leverage the FUSE [6] framework on Linux to provide cross platform support for programming languages and runtime environments that can run on multiple operating systems. We will also add support for multi-user scenarios with various security and scalability options.

References

1. Callback File System – http://www.eldos.com/cbfs/.
2. Condor project – http://www.cs.wisc.edu/condor/.
3. D4Science: Distributed collaboratories Infrastructure on Grid ENabled Technology 4 Science – http://www.d4science.eu/.
4. Dokan file system – http://dokan-dev.net/en/.
5. EGEE: Enabling Grids for E-Science in Europe – http://public.eu-egee.org/.
6. Filesystem in Userspace – http://fuse.sourceforge.net/.
7. GridFTP – http://globus.org/toolkit/data/gridftp/.
8. MacFUSE – http://code.google.com/p/macfuse/.
9. NorduGrid: Nordic Testbed for Wide Area Computing and Data Handling – http://www.nordugrid.org/.
10. The Globus toolkit – http://www.globus.org/toolkit/.
11. E.-D. A.E, A. H.A, and A. A.A. A pure peer-to-peer desktop grid framework with efficient fault tolerance. In *ICCES'07*, 2007.
12. W. Allcock, J. Bresnahan, R. Kettimuthu, and J. Link. The globus extensible input/output system (xio): A protocol independent io system for the grid. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 4*, page 179.1, Washington, DC, USA, 2005. IEEE Computer Society.
13. A. Azab. Himan: A pure peer-to-peer computational grid framework, 2010.
14. J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Explicit control in a batch-aware distributed file system, March 2004.
15. M. Ernst, P. Fuhrmann, M. Gasthuber, T. Mkrtchyan, and C. W. dCache. A distributed storage data caching system. In *Computing in High Energy Physics*, 2001.
16. M. B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *fourteenth ACM symposium on Operating systems principles*, 1994.
17. S. Klous, J. Frey, S.-C. Son, D. Thain, A. Roy, M. Livny, and J. van den Brand. Transparent access to grid resources for user software. *Concurrency Computat.: Pract. Exper*, 18, 2006.
18. H. Lederer, G. J. Pringle, D. Girou, M.-A. Hermanns, and G. Erbacci. Deisa: Extreme computing in an advanced supercomputing environment, 2007.
19. M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. Socks protocol version 5, 1996.

20. B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. B. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tools. *IEEE Computer*, 28(11), 1995.
21. R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The use of name spaces in Plan 9. *SIGOPS Oper. Syst. Rev.*, 27(2):72–76, 1993.
22. G. R. Grid3 : An application grid laboratory for science. In *Computing in High Energy Physics and Nuclear Physics*, 2004.
23. S. S and L. M. Recovering internet symmetry in distributed computing. In *CCGrid*, 2003.
24. D. Thain and M. Livny. Multiple bypass: Interposition agents for distributed computing. *The Journal of Cluster Computing*, 4(3), 2001.
25. D. Thain and M. Livny. Parrot: Transparent user-level middleware for data-intensive computing. *Scalable Computing: Practice and Experience*, 6(3), 2005.