DataLight: data transfer and logging of large output applications in Grid environments

Paulo Abreu[1], Ricardo Fonseca[1,2], Luís O. Silva[1]

[1] GoLP, Instituto de Plasmas e Fusão Nuclear, Instituto Superior Técnico, Lisboa, Portugal
paulotex@ist.utl.pt

[2] Departamento de Ciências e Tecnologias da Informação, Instituto Superior de Ciências do Trabalho e da Empresa, Lisboa, Portugal
ricardo.fonseca@iscte.pt

Abstract. In this paper we present a powerful and lightweight library called DataLight that handles the transfer of large data file sets from the local storage of the Worker Node to an gLite Storage Element. The library is very easy to integrate into non-Grid/legacy code and it is flexible enough to allow for overlapping of several data transfers and computation. It has the ability to send messages to a Web Portal while the application is running, so that a user can login to the Portal and receive immediate updates. In particular the user can start post-processing the produced data while the application is still running. We present the application details both of the library and the Portal and discuss some relevant security issues.

Keywords: web portal, large data sets, Grid migration, data transfer.

1 Introduction

The need for a large-scale distributed data storage and management system has been rising in the last forty years at least as fast as the need for computational power. In fact, the recently formed European Grid Infrastructure (EGI) has its roots partially in the European Union funded DataGrid project [1], aimed at sharing "large-scale databases, from hundreds of TeraBytes to PetaBytes, across widely distributed scientific communities." This makes the EGI highly optimized for dealing with both data intensive and computational intensive applications, since two of its main purposes are to offer high computing power to applications and high storage and replication facilities for data. However, this leads to a certain dominance of Grid applications that *require* large amount of data as input (like the ATLAS or CMS experiments), against Grid applications that *produce* large amount of data as output.

A very common type of high-performance computing (HPC) algorithm is one that produces very large data files as output. An example is the class of algorithms used in physics simulations, where the motion and interaction of several thousands or millions of particles are simulated. Codes like this (e.g., PIC [2]) require a relatively small amount of data as input (e.g., ~ 1 kB) but can produce a huge amount

of output (e.g., up to 1 TB). In our particular case, we develop, maintain and deploy two massively parallel plasma simulation codes (Osiris [3] and dHybrid [4]), which have been tuned for HPC systems ranging from one to several thousands of processors. These codes, besides being good candidates for application deployment in the EGI Grid, also challenge the current migration infrastructure, since they usually produce large data sets (several hundreds gigabytes of data) as output for post-processing (e.g., visualization).

Hence it is important to make these large amount of data produced by such codes available as quickly as possible, ideally still at runtime, so that the user can have almost immediate access to preliminary results. It is also likely that if the application is writing the results locally on the Worker Node (WN), this node might not have enough storage for the complete output data, specially when it is in the order of several hundreds of GB.

In this paper, we present an improvement and expansion of a previous tool that we have developed [5], which we now call DataLight. In its current form, DataLight is a library that eases the deployment on the EGI of applications that output large amounts of data, specially in the case where that data should be made available as soon as it is produced. It allows for multi-threaded transfers of data, thus taking advantage of current fat nodes (some cores in a computing node can be used for the transfer threads while the rest is doing the calculations and generating the data), and it automatically updates the status of the program by sending messages to a Web portal. This allows for an immediate feedback to the user, that can start post-processing the data while the program is still running.

This paper is organized as follows: in Section 2 we present an overview of previous work that has been done in this area, in particular in the available gLite tools for data transfer and in the previous tool that we have developed; in Section 3 we present a general description of the algorithm developed for the library and give an overview of the solutions we found for the user interface; in Section 4 we present the Web portal implementation and the security issues that had to be solved; finally, in Section 5 we conclude and point to some directions for future development.

2 Previous work

A great amount of work has been done on the ability to replicate and transfer large data sets on the Grid [6–8]. In fact, one of the main features of the EGI middleware (gLite) is its emphasis on data handling, namely on its replication and accessibility. However producing large data sets as result of simulations running on the EGI poses an interesting problem for application development. On one hand, gLite offers two high-level APIs for data management (GFAL and lcg_util [9]) which are well suited for the reliable transfer, replication and management of data on the Grid, and one low-level API for data transfer submissions (FTS [10]), which is suited for point to point movement of physical files. On the other hand, such data management operations should occur still *during* simulation time, not just after it, such that the generated data is made generally available as soon as possible and the Worker Nodes (WN) are never filled with too much output results. This

requirement can lead to performance degradation (either at the application level or at the WN level) due to the overhead and slowness of network transfers when compared to local storage access.

In [5] we explained in detail the reason for our choice of the lcg_util API. In an overview, we need to deal with complete files (unlike GFAL, which uses POSIX to create, write to, and read from, files on the Grid), we need a higher level abstraction to data management (unlike FTS), and we would like to base our library in a stable and powerfull data transfer layer like GridFTP [11, 12]. Although other tools have been proposed for file transfers on the Grid (like [13]), we found that a GridFTP based tool had all the requirements of stability, reliability and performance that we needed. Also, GridFTP is one of the most widely used middelware for data transfer on Grid systems, so building our library on top of a software stack based on GridFTP is a first guarantee for portability.

3 DataLight Implementation

Figure 1 shows a simplified fluxogram of the algorithm in DataLight. It is based in [5], but with several important additions, that will be explained in the following sections.

Our general goal was to develop a tool that would ease the deployment on the Grid of applications that output large amounts of data, specially in the case where that data should be made available as soon as it is produced. To this end, we have improved on a previous implementation (described in [5]), adding several configuration capabilities and an interaction with a Web portal.

The top left of Figure 1 represents the main application that uses DataLight. It mainly consists of a simulation cycle followed by a write of the result data to local storage; these two steps are repeated until the complete simulation finishes. Each local write operation is followed by a call to the DataLight function (`write_remote`), that adds the file to a waiting queue of files waiting to be transferred.

The first time the main app calls this function, it launches also a queue manager thread in DataLight (right hand side of Figure 1), which is responsible for managing the waiting queue. This queue manager dispatches the files on the waiting queue list in a FIFO order to a transfer queue. It checks if there is enough transfer slots available (these are bandwidth and system dependent and can be defined by the user) and, for each file that is moved from the waiting queue to the transfer queue, it initiates a transfer thread (bottom left of Figure 1).

Each transfer thread uses a simple algorithm represented on the bottom left of Figure 1. It is up to each transfer thread to remove the file reference from the transfer queue and to remove the local file as soon as the transfer finishes.

Finally, as the complete simulation finishes, the application sends a signal to the queue manager thread in DataLight by calling the function `write_finished`. This forces the queue manager to wait until all queues are empty and then quit, thus allowing the application to finish.

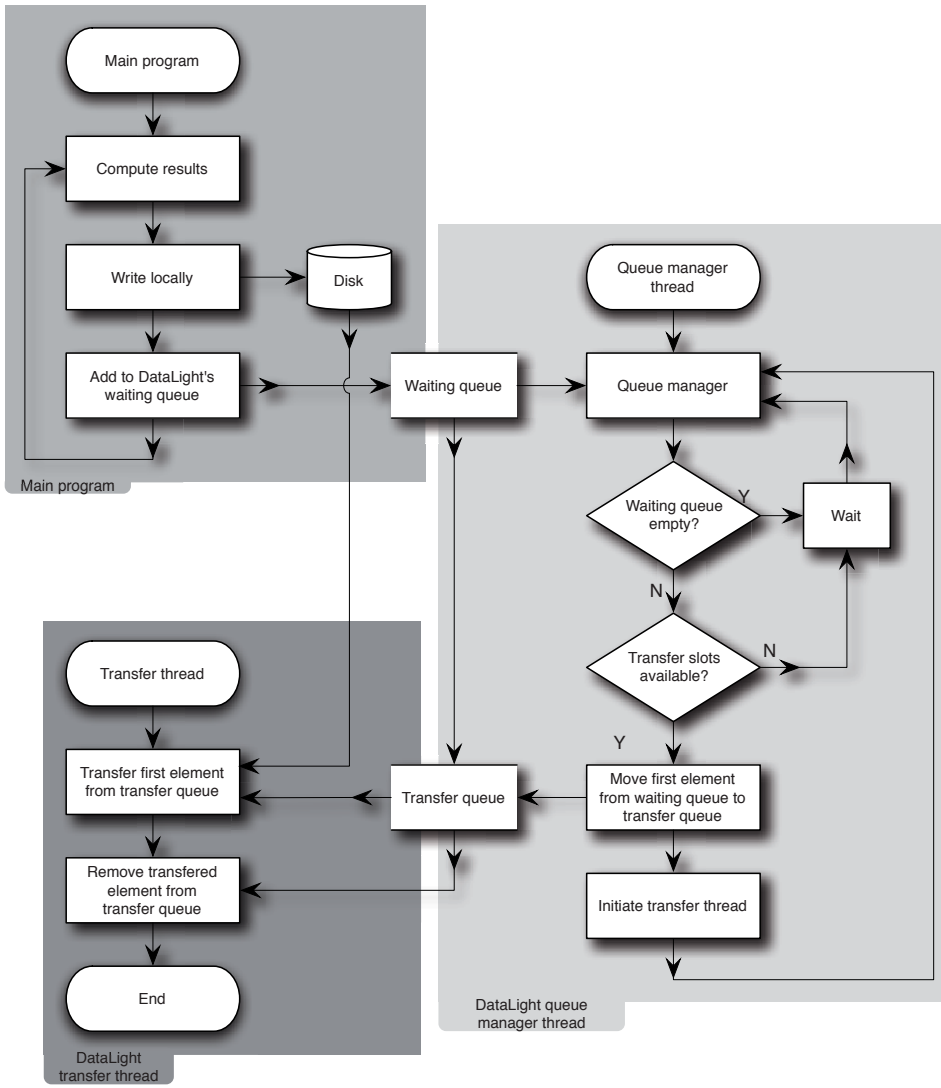

Fig. 1. A simplified fluxogram of the DataLight library (left and bottom right), together with the application that might use it (top left).

3.1 User interface

The two functions referred to in the previous paragraphs (`write_remote` and `write_finished`) are the only exported functions of the DataLight library. This allows for a minimal code intervention in the application that wishes to write files to the Grid. This was one of the main goals of the library, since it is expected to be deployed with applications that were built without the Grid in mind. This saves the cumbersome and error prone change of (possibly complicated) output routines.

Several aspects of the behavior of DataLight can be controlled by environment variables. We consider this approach to be more flexible than passing parameters directly to the called function. On one side, it keeps the API minimal and hence avoids extensive legacy application code editing. On the other side, environment variables can be simply set in the JDL file the define the Grid job, thus allowing for a general configuration of the application and not of a particular function call.

All DataLight variables start with the prefix `DL_` and the library tries to use sensible values if they are not set. Here is the complete list, together with a description and the default value:

`DL_DEBUG` It is a numerical value that sets the verbosity level of the library. At the moment it uses two values: '0' for no extra messages (this is the default behavior if this variable is not set) and '1' for extra logging. Any other integer value is handled as a '1'. Any non integer value is handled as a '0'.

`DL_VO` It is a string the contains the Virtual Organization this job belongs to. It is used to create a correct Logical File Name for the each file that is transferred. If not set, the environment variable `LCG_GFAL_VO` is used. If this variable is not set, an error message is send and this value is empty.

`DL_JOBID` A string that uniquely identifies this job. This will be used in the portal to collect all messages from the same job. Normally, it should be not set, since DataLight uses the value from `GLITE_WMS_JOBID`. If this variable is not set, it uses `EDG_WL_JOBID`. Finally, if neither variables are set, DataLight issues a warning and uses the `date()` function. In this case, it is worth noting that this value will stay the same during all the time that the application is running.

`DL_SE` This is a string that contains the FQDN of the Storage Element where the files should be transferred to. If it is not set, than its value is copied from the environment variable `VO_${DL_VO}_DEFAULT_SE` (an automatic conversion of `DL_VO` to upper case is done). If this variable is not set, an error message is send and the program continues, but without data transfer.

`DL_NTHREADS` This variable is a numerical value that sets the number of simultaneous file transfer threads (represented in the bottom left of Figure 1). Default value is '1'. Negative values or non integer values act like a '0', and no file transfer takes place. This is useful if one wishes to use just the Portal capabilities of DataLight to have an updated situation of the running application, without the need to actually transfer any data.

`DL_NB` This is equivalent to the `nbstreams` parameter of the `lcg_cr` command. It specifies the number of parallel streams per file transfer thread. The default value is '1'.

`DL_USER` This is a string variable that should be set to the user name in the UI that runs the Portal (more on this later). There is no default value. If the variable is not set, it is most likely that DataLight will not be able to contact the Portal.

`DL_ROOT` This variable is a string that defines the root path for creating the correct Logical File Name for the each file that is transferred. Usually, it should be set to the user's CN of the X509 certificate. In the LFN, the local path of the file, as passed to the function `write_remote`, is reproduced. As a complete example (also using the variable `DL_VO`), let us assume that files are locally written in the directory `output`, that the VO is "test" and that `DL_ROOT` is set to "`paulo_abreu`". In that case, each `write_remote` function call will have the format `write_remote('output/...')` and the complete LFN for each transferred file will be:

```
/grid/test/paulo_abreu/output/...
```

where `...` represents the name of each file. There is no default value.

`DL_PORTAL` This string has the FQDN of the Web portal page that is to be contacted with messages from the DataLight library. The next section has more information on this issue.

Most of this variables do not need to be set, since the default values do what is to be expected. The only recommended variables to be set are `DL_ROOT`, so that LFN entries are locally kept in directories related to the user, and `DL_PORTAL`, which points to the Web portal to be updated while the job is running. Without this last variable, the user will only be able to access the transferred files by consulting the `stdout` file, after the job has finished, instead of having an immediate access to them during the job, through the Web portal.

4 Portal Implementation

Another important feature that we have implemented in this version of DataLight is the ability to send HTTP POST messages while it is running. In the current version of the library, we have implemented messages related with file transfer (success, failure, retry, ...), but any kind of message can easily be added.

These messages are sent to a Web page URL defined in the environment variable `DL_PORTAL`. Obligatory fields for each message is its identification (called ID), the user this messages belongs to (called Owner), and some text (called Log). The ID is the environment variable `DL_JOBID`, the Owner corresponds to the `DL_USER` variable and should be the name of a user registered in the system the Portal is running on (usually, the UI), and the Log is any text that the library finds

necessary to send to the user. For example, each time a file is written to the SE and removed from the local WN, DataLight sends a message with the LFN of the transferred file in the Log.

The Portal itself is written mainly in PHP and interfaces with a MySQL database. Each message is added as an entry to the database and the current date and time added to it.

At the moment, the main use of the Portal is to register the transfer of files from the WN to the SE using DataLight. By logging into the Portal, the user can check the status of the job and has access to the LFN of each file that was transferred. Then, with the usual gLite tools (for example, `lcg-cp`) the user can get the file from the SE to the UI for post-processing, while the simulation is still running.

An important issue to note is that each WN must be allowed to have an outbound network connection. This can be achieved be setting the `GlueHostNetworkAdapterOutboundIP` attribute to true.

4.1 Security issues

We have identified two points where security of the Portal can be an issue. The first point is in receiving the HTTP POST messages: without proper care, it is easy to flood the Portal with false messages (DoS attack). The second point is in accessing the Portal to receive information about jobs: only the owner of a job should be allowed to view its messages.

To handle both issues, the Portal in its current form is deployed on the UI of a site and only allows secure connections from users registered on that UI. When a user authenticates with the Portal, it only has access to the messages received that have the Owner field set to the correct user name. All other entries are not accessible. Also, the Portal silently drops any message that has an owner field that does not correspond to a user. This limits the possibilities of misusing of the Portal.

5 Conclusions and future work

One of the main strengths of DataLight is its simplicity. It allows for the overlapping of computation and data transfers with a minimal effort to the developer of the non-Grid application. By implementing the corresponding Portal, the generated files are readily available to the users while the application is running. For applications that require long computational times (several hours or days) and that write hundreds of files, this is another major feature of the library.

However, we consider the DataLight Portal in its current form to be at an initial stage. An important next step will be to offer an user interface in the Portal to control the launching of jobs: after login, the user specifies the input files and parameters, and the Portal automatically generates the JDL file and launches the job. The allows for an increased security, since the Portal has access to the WMS job ID from the beginning, and can be configured to only accept remote messages

from running jobs with valid IDs. Furthermore, this also lifts the necessity of hosting the Portal in the UI: any user with valid Grid credentials can apply for a login to the Portal.

The final step will be to implement some post-processing capabilities in the Portal. Getting the files from the SE to the UI is trivial, if the user has the corresponding LFN. But it is also possible to implement *in-situ* post-processing (with the files staying in the SE) using GFAL and a visualization tool with a browser interface (for example, Vtk [14] or VisIt [15] with Java bindings).

Acknowledgements

This work was partially supported by grant GRID/GRI/81800/2006 from FCT, Portugal.

References

1. The CERN DataGrid Project. `http://www.cern.ch/grid/`.
2. C. K. Birdsall and Langdon. *Plasma Physics via Computer Simulation (Series on Plasma Physics)*. Taylor & Francis, January 1991.
3. R. A. Fonseca, L. O. Silva, F. S. Tsung, V. K. Decyk, W. Lu, C. Ren, W. B. Mori, S. Deng, S. Lee, T. Katsouleas, and J. C. Adam. OSIRIS: A three-dimensional, fully relativistic particle in cell code for modeling plasma based accelerators. In Sloot, P. and Tan, C. J. K. and Dongarra, J. J. and Hoekstra, A. G., editor, *Computational Science-ICCS 2002, PT III, Proceedings*, volume 2331 of *Lecture Notes in Computer Science*, pages 342–351. Springer-Verlag Berlin, 2002.
4. L. Gargaté, R. Bingham, R. A. Fonseca, and L. O. Silva. dHybrid: A massively parallel code for hybrid simulations of space plasmas. *Computer Physics Communications*, 176(6):419–425, 2007.
5. P. Abreu, R. A. Fonseca, and L. O. Silva. Migrating large output applications to Grid environments: a simple library for threaded transfers with gLite. In *Ibergrid 2008 — 2nd Iberian Grid Infrastructure Conference Proceedings*. Netbiblo, 2008.
6. Mehnaz Hafeez, Asad Samar, and Heinz Stockinger. A Data Grid Prototype for Distributed Data Production in CMS. In *7th International Workshop on Advanced Computing and Analysis Techniques in Physics Research*, 2000.
7. Mehmet Balman and Tevfik Kosar. Data scheduling for large scale distributed applications. In *Proceedings of the 5th ICEIS Doctoral Consortium, In conjunction with the International Conference on Enterprise Information Systems (ICEIS?07*, 2007.
8. Caitriana Mairi and Macleod Nicholson. *File Management for HEP Data Grids*. PhD thesis, University of Glasgow, 2006.
9. S. Burke, S. Campana, A. Peris, F. Donno, P. Lorenzo, R. Santinelli, and A. Sciabà. *gLite 3 User Guide*, 2009.
10. Peter Kunszt. *File Transfer Service User Guide*, 2005. `https://edms.cern.ch/document/591792/`.
11. The Globus Project. GridFTP: Universal Data Transfer for the Grid. White Paper, September 2000. `http://globus.org/toolkit/docs/2.4/datagrid/deliverables/C2WPdraft3.pdf`.
12. B. Radic, V. Kajic, and E. Imamagic. Optimization of Data Transfer for Grid Using GridFTP. In *29th International Conference on Information Technology Interfaces, 2007.*, pages 709 –715, June 2007.

13. Jiazeng Wang and Linpeng Huang. Intelligent file transfer protocol for grid environment. In Wu Zhang, Weiqin Tong, Zhangxin Chen, and Roland Glowinski, editors, *Current Trends in High Performance Computing and Its Applications*, pages 469–476. Springer Berlin Heidelberg, 2005.
14. The Visualization Toolkit. `http://www.vtk.org`.
15. VisIt: Software that Delivers Parallel Interactive Visualization. `https://wci.llnl.gov/codes/visit/`.