Standardized multi-protocol data management for grid and cloud GridRPC frameworks

Yves Caniou, Hadrien Croubois, Gaël Le Mahec

Standardized multi-protocol data management for grid and cloud GridRPC frameworks

Yves Caniou[1], Hadrien Croubois[2], and Gaël le Mahec[3]

[1] Université de Lyon, JFLI CNRS, Japan,
Yves.Caniou@ens-lyon.fr
[2] Université de Lyon, ÉNS Lyon
Hadrien.Croubois@ens-lyon.fr
[3] Université de Picardie Jules Verne, MIS Laboratory, France,
Gael.Le.Mahec@u-picardie.fr

Abstract. GridRPC is an international standard of the Open Grid Forum defining an API designed to allow applications to be submitted in a seamless way on large scale, heterogeneous and geographically distributed computing platforms. First versions of the standard did not take into account any data management feature. Data were parameters of the Remote Procedure calls, without any possibility to prefetch them, to use persistence, replication, external sources, *etc.* , and making GridRPC codes middleware dependent. The data extension of the standard introduced a short set of functions and data structures to complete the API with simple but powerful data management features. In this paper, we present a modular and extensible implementation of both APIs, which needs only a few developments to be usable with any middleware relying on RPC, and which provides access to numerous and easy to extend protocols and data middleware to access data. Gaining data management functions, it introduces interesting potentiality for optimization that such an approach would provide to large scale applications.

1 Introduction

Many applications use RPC-like mechanisms to distribute computations over nodes of clusters and supercomputers composing some distributed systems like a grid, a cloud, or both (now referred as sky computing). Combined with connections to huge databases, they more or less transparently provide scientists with the possibility to focus on their core thematic, giving them more time to deal with data analysis, without dealing with the underlying complexity of all the different mechanisms involved into job and data management. More lately applications even directly couple analysis, graphical representations and such, making platform management only a part of their project, whose actions are generally available through some web site. And surprisingly, when considering a new area, a new platform, new independent pieces of software are often developed instead of using previous work, software or standardized APIs.

The Open Grid Forum standard defining the GridRPC paradigm, namely Remote Procedure Call over the Grid, has been published in 2007, benefiting

from 10 years of experience by their respective authors. Simple and easy to use, it has been completed with a standardized data extension only recently. This extension to the native API proposes to expert users to easily handle remote data and to optimize distributed applications with prefetch, migration or replication of possibly distant data using multiple asynchronous transfers together with remote procedure calls on available distributed computing resources.

Based on preliminary experiments[1, 5], applications also benefit from multi-administration sites resources managed by multi-middleware (inherent to interoperability provided with the implementation of the API data extension) and target not only traditional Grids but any distributed platform possibly composed of resources from the Cloud [6].

In an attempt to simplify and develop interoperability, and to unify previous works, we propose here a library managing *both* GridRPC and GridRPC Data Management APIs. We present an overview of the project architecture, designed with a very modular prospect, relying on middleware and data manager modules but also bringing inner data manager capabilities and transfer protocols. Having in mind not to go too much into details, we highlight here some of its features, such as the asynchronous requests management and the transfer management, which involves mapping and scheduling aspects: there is interesting potentiality for optimization at the data operation level, with scheduling to reduce the completion time of a data operation when several sources and several destinations are provided but not necessarily interconnected; and at the workflow/dataflow level to reduce any [sub part of an] application graph. At the moment, the library provides modules for the grid middleware DIET and Ninf , and data manager modules for projects and protocols like `Dagda`, `iRods`, `webdav` (used for web-based repositories like dropbox, owncloud), `ftp` and `rsync`.

The rest of the paper is organized as follows: next section explains the motivations behind the GridRPC DM API and some related work. Section 3 presents the global design of the implementation, the different issues that the API leads to and their solution. Section 4 presents some validation experiments and after explaining some future work directions, we conclude in Section 6.

2 State of the Art

2.1 The GridRPC Data Management API, Summary

The GridRPC DM API [2] introduces the concept of data handle and with it, several GridRPC data types to provide standardized information, for example lists of input and output URIs to give the locations of respectively source and destination [remote] data, with the according protocols to access it at the considered location). It also defines mode managements for a client to characterize the persistence of the data in the system, *etc.* All actions (initializing, transferring, waiting for completion of asynchronous transfers, *etc.*) are provided with only 12 functions.

This standard answers at the API level to issues related to **feasibility** of the computation by decoupling the data from its locations and from protocols

to access it; to **performance** using different sources and protocols to access a remote data, providing explicit data management with the possibility to prefetch and to migrate data, as well as the possibility to rely on some smart middleware to transparently handle data management; and to **extensibility** by providing containers of data. It also solves **portability**, making GridRPC codes portable from one middleware to another.

2.2 Related Work

Similar works can address some data management issues in the GridRPC but only separately and without integration into remote procedure call: one can store data on a distributed file system like GlusterFS[4] or GFarm [9] to deal with automatic replication; OmniRPC introduced omniStorage [7] as a Data Management layer relying on several Data Managers such as GFarm and Bittorrent. It aims to provide data sharing patterns (worker to worker, broadcast and all-exchange) to optimize communications between a set of resources, but needs knowledge on the topology and middleware deployment to be useful; DIET also introduced its own data managers (DTM and Dagda [3, 4]), which focus on both user explicit data management and persistence of data across the resources, with transparent migrations and replications.

At a higher level, Stork [8] is a batch scheduler specialized in data placement and data movement. If the transfer protocol specified in the job description file fails for some reason, Stork can automatically switch to any alternative protocol available between the same source and the destination hosts and complete the transfer; Galaxy[5] is a web interface written in python allowing on-line design of task workflows. Galaxy focuses mainly on bioinformatics but could be used for all type of applications relying on workflow execution. By default Galaxy is configured to execute application on its host server but can use the OGF DRMAA API to distribute computations on remote servers. Data can only be transferred as files. On the contrary of classical RPC, there is no simple way to upload data directly on the application memory address space. Moreover, the GridRPC API modularity allows to combine simplicity of such data management systems and tunability by choosing where and when data are transfered.

By using standardized GridRPC code with our implementation and its corresponding modules, it should be possible to benefit at a upper layer from previous works, gaining in portability and interoperability with middleware and data managers, which in turn provides access to a potentially larger set of resources and architectures.

3 Implementation: architecture and features

We present in this section the system underlying our implementation of the GridRPC and GridRPC Data Management standards. We highlight the features

[4] `http://www.gluster.org/`
[5] `http://galaxyproject.org/`

of the library, its data management capabilities as well as scheduling possibilities between and for each data operation, *i.e.,* the set of all transfers requested between the URIs provided as sources and destinations for the same data.
The library is developed in C++ and C, using internally `boost`, and `cmake` to build the project. It is freely available from a sourceforge repository: `http://sourceforge.net/projects/gridrpcdm/`.

3.1 Modularity of the solution

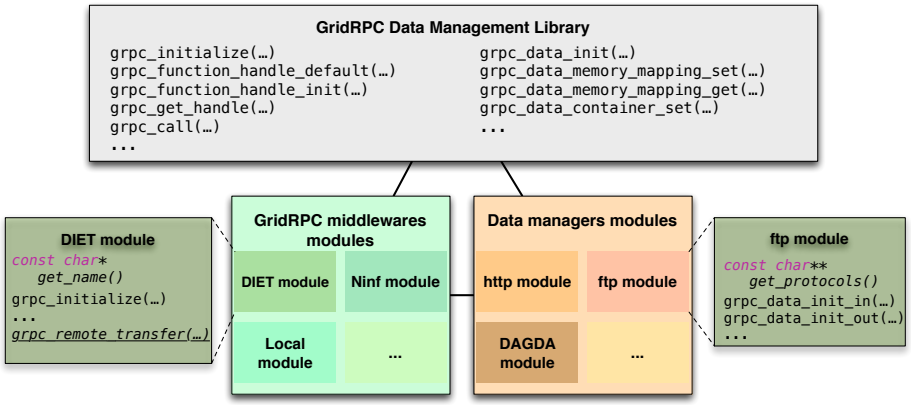

Fig. 1. A very modular implementation

The proposed implementation can be viewed as a *meta-implementation*) of the APIs (see Figure 1) since it provides the two GridRPC APIs, adding some seamless mechanisms for performance (scheduling etc.) in a middleware and protocol "agnostic" manner. The library does not interact directly with the middleware nor the data storage servers. It proposes a fully interoperable API for any middleware and protocol/data manager with only very few specific developments of simple modules. The module developers do not have to take care about which data transfer protocol is available, like the data manager module developpers do not have to care about which middleware is used to call remote procedures. To do so, different interfaces are provided by the library:

- The client application interface: the external client API. Clients can use the API directly without any knowledge about the underlying GridRPC middleware. However, by adding a prefix to the service name, users can force the library to use a specific middleware (*e.g.,* `"DIET:matmul"` and `"Ninf:matmul"` select respectively DIET and Ninf-G for the `"matmul"` service).
- The Services interface: it is a subset of the client API with some additional utility functions facilitating servers conversions from standard GridRPC servers to GridRPC Data Management servers.

– The Modules interface: the library defines a set of functions that should be exposed by the module to extend the library capacities.

Integrating a new middleware requires to fulfill a set of 10 main functions and one optional. Most of them are just type conversions functions from the existing middleware data-type to the new GridRPC data-types. The most complex function of a middleware module is `grpc_remote_transfer()` which initiates a transfer from a remote host to another remote host. A default implementation is included in the library relying on a middleware service call: the module developpers have just to implement this simple service using the library transfer capabilities on the server side.

Remarks:

– To avoid "name conflicts" between existing GridRPC implementations and the new definitions of the library, definitions in the library headers files are automatically prefixed when needed, allowing an easy reuse of the existing functions without name-clashing at the compilation step.
– Note on asynchronous calls: they are internally managed by the library from synchronous calls to middleware. However, middleware functions must be reentrant for a safe asynchronous use.

At the moment, the modules for the DIET and Ninf GridRPC middleware are available.

Integrating a new data manager module requires to provide 4 functions: 2 initialization functions corresponding to input and output data, which can most of the time be left empty; and 2 transfer functions to get and put a data. They are generally wrappers of existing transfer protocol libraries (*e.g.,* `libcurl` for `http` and `ftp`).
At the moment, the library implements the data manager modules for `rsync` and `scp`, using the shell commands; 2) `iRODS`, using the shell command (the library is only available for Java and PHP); `webdav`, to access `Owncloud` and `Dropbox` servers. It uses the `neon` library and; `curl`, to access `http` and `ftp` via the `curl` library.

Module initialization The library global initialization process reads the global configuration file to determine which module should be dynamically loaded at execution time, where to find it, and some parameters available for each module in its own separate section.

The initialization function of each module is then processed sequentially, passing the arguments of the module specific configuration, and potentially reading more parameters in the deployed module-specific configuration file.

3.2 Asynchronicity management

We call a request the inner action managed in the library: they correspond to API calls for remote procedure, API calls for a transfer or a group of transfers involving a unique data. For example when one source and several destinations are provided as input and ouput URIs, several transfers are involved in group to provide the unique API transfer call. All requests are managed the same way by the request controller: this entity registers each of them during their initialization, and with the help of threads and semaphores it limits their number and immediately knows the idendity of a request that completes without active wait. Some additional dependency information is also recorded with each request, and thus a hierarchy of requests (the link being the temporal dependency) can be built. It is used to express the concept of a group of requests reported above in the transfer example, but it is also a powerful way to handle waits for one or a group of asynchronous remote procedure calls as well.

Requests are managed with a priority system, which has been instantiated in the current implementation with a queue managed with a FIFO algorithm and a limitation on the number of parallel threads executed at a given moment: There is not much more that can be done at the moment: since there is no dependency information between data transfers operated at the API level, one cannot try any optimization between requests that do not belong to the same group because it could generate inconsistency in data or failure. However coupled with a system that handles workflow/dataflow, some meta-scheduling over available GridRPC middleware and data managers may be performed.

3.3 Data manager capabilities

The library does not only operate with underlying data transfer projects. It must provide the data persistence as defined in the API, integrate the possibility to communicate in-memory data (which possibly avoids at least one copy to disk), and make the junction between different locations where the data is available, and the protocols with which one can access them. The latter induces possible hidden (automatic and mandatory) copies and scheduling for the data to be transferred to all requested destinations.

Data persistence The GridRPC Data Management API defines numerous persistence modes: the data can be volatile, *i.e.,* there is no special requirement on its management and this can be considered as the default mode; it can be strictly volatile, meaning that the library has to provide means to remove the data from the platform after a computation (thus some protocols and data managers cannot be used); when defined as sticky, the data or a copy must be kept on the location where the client requests are executed; if unique sticky, no replication nor migration can be performed; finally, the client can also request the library to transparently manage prefetch, replication and migration of data. Then, by also handling procedure calls the library can perform some scheduling in order

to reduce some metrics. At the moment persistent data are managed through DAGDA .

The `memory` protocol Each data is referenced by a given set of specific URIs, providing the transfer protocol or the underlying data manager to use (for example `http` or `dagda`). But when trying to get performance, on linear algebra computation for example, there is a need to keep data in memory and avoid file transfers. The GridRPC Data Management API foresaw this kind of use and introduced the `memory` protocol. In addition to this protocol management, our implementation lets a client (or the library itself) use URIs with query and fragment. This leads to possible evolution for improvements (see after) and to manage more data managers (like P2P middleware that initiate torrents with specific files).

Implementing the `memory` protocol means that the library has to use GridRPC middleware inner data manager which can hopefully communicate between its own components to achieve such a need. But when a data is in memory and has to be transferred either on another GridRPC middleware components or on a storage server, it has to be written to a file and then be manipulated (transferred and possibly handled remotely) to be in the requested status. This part uses (de)serialization functions, defined by the GridRPC Data Management API, partly relying on tools provided by the `boost` library. But that maybe shows an unclear part of the API: the protocol to use in that specific case to manipulate the file is not precised. In our current implementation, the protocol is static and is read from the middleware configuration file at initialization time.

But if going a bit further than the API, we can use the query part of the URI. Indeed considering a data available in memory, the API does not provide a mean to know which protocol(s) can be used to send or to receive it since the URI would be similar to `memory://graal.ens-lyon.fr/matrixA`. In ongoing work, our library is going to explore what can be done with specifying protocols within the URI query part, *e.g.,* `?protocol=rsync?protocol=webdav`.

Scheduling for implicit and explicit data transfers Data transfers are operated 1) when data participate to a remote procedure call. They are in that case implicit or automatic, and; 2) can be explicitly requested by a client with a call to `grpc_data_transfer()`.

Implicit and automatic transfers: When a remote procedure call is performed, meta-data are serialized and transferred to the distant service. They contain sets of URIs which may lead to additional transfers before and after the service execution (Fig 2).

- If one of the input URI refers to a memory or file data on the client, the data must be available to the service before its execution so that it can remotely access it.
- If one of the ouput URI refers to a memory or file data on the client, the service must have made it available and the client must get it.

Explicit data transfers: Several transfers are operated by `grpc_data_transfer()`, *i.e.,* a call to an explicit transfer operation: the data should be present in all locations set in the input URIs list, and must be present in all locations set in the output URIs list. This can be treated with a sequential set of transfers from one given source to each destination for example, but that would be inefficient. In addition, it is not mandatory that all participants are directly interconnected (either by network or by protocol) and transfers may have to be scheduled to make the whole operations possible (destinations of completed transfers being considered as potential sources). The library also makes possible to delegate transfers on all GridRPC servers that offers some library specific service. Hence transfers can be distributed over nodes to reduce the bandwidth impact, and/or to try to reduce the transfer operation completion date for example.

In order to build a schedule in our implementation (made by the dispatcher, Fig 3), we list the nodes that can participate to a transfer operation: To our benefit, since the library contains middleware modules, it can also rely on underlying GridRPC middleware to potentially add relay servers to [remotely] distribute the transfer load or a part of it. To discover those middleware nodes, the library provides an `echo` service, that must be deployed, *i.e.,* registered in the GridRPC server capabilities (at the moment, only middleware nodes with the `memory` protocol available are considered. If the service is not deployed, the node is simply not considered as a possible relay).

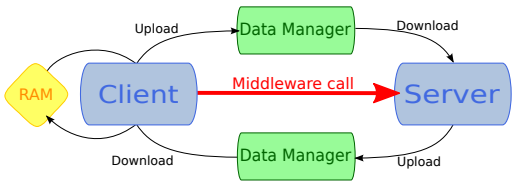

Fig. 2. Automatic transfers during a GridRPC call

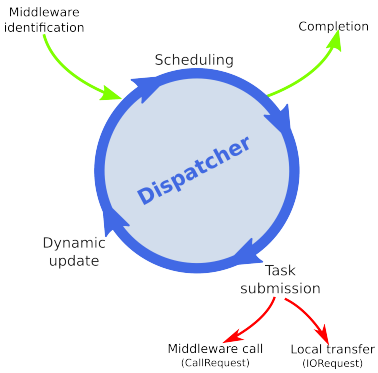

Fig. 3. Dispatcher's cycle

Then URIs are sorted: Local, Middleware node or Storage server; and the dispatcher uses a Round-Robin algorithm to build and launch every one-to-one transfer according to the sorted list below: the list describes *by priority of action*, matching one input URI to one output URI depending on their nature (Local, Middleware or Storage), the action undertaken to manage the corresponding transfer. As seen in Sec. 3.2, transfers at the same time are possibly limited in number, they are monitored with an effective and sufficient semaphore mechanism, and a completion leads to a dynamic update of the set of input URIs.

The above algorithm loops until all transfers to ouput locations are done. In case of failure due to an unresponsive input middleware, the middleware is not considered anymore in the next scheduling/mapping cycle.

L-S: The transfer is initiated locally.
M-S: The transfer is processed through a call to the remote transfer service.
S-L: The transfer is initiated locally.
S-M: The transfer is performed through a call to the distant transfer service.
L-L: The transfer is initiated and performed locally.
L-M: The library makes the local data available via the GridRPC middleware inner data manager whose remote counter part will download afterwards.
M-L: The remote middleware is being asked to make the data available, this data is then downloaded by the GridRPC middleware.
M-M: The source middleware is asked to make the data available so that the destination middleware can download it when needed.
S-S: The library first tries to invoke a remote service on the destination server to initiate the transfer. If the transfer fails, data is downloaded on the library client, then transferred to the destination server. If there is no available protocol to proceed to such transfers, the call fails returning an error code.

4 Experimental results

4.1 Multi-protocol and dispatcher scheduling/mapping validation

Table 1 lists the experiment deployment. We used 3 computing resources, 2 in Japan and 1 in France, on which we deployed `iRODS` and `ssh` servers, and DIET components: a client, a dietAgent (the registry), and a server (matrix addition) written with the GridRPC APIs requirements together with our library. Two matrices are defined with a list of input URIs depending on the running test, described hereafter.

Machine (location)	Services	Data (protocol)
Arcterix (JFLI - Japon)	dietAgent, client, `sshd`	matA (`ssh`)
yume (JFLI - Japon)	service '`+`', `iRODS`, `sshd`	matA, matB (`ssh`)
graal (Éns-Lyon - France)	`sshd`	matB (`ssh`)

Table 1. Resources involved in Experiment 1

There are four tests, built with the scenario of getting the two matrices through `ssh`, performing the addition, and uploading the result to an `iRODS` server (here locally):

– Remote/Remote: the client does not include the URIs concerning the host *yume* in the input list used for the remote call.

– Remote/Local: the client does not use the URI concerning matA on *yume* in the input list used for the remote call.
– Local/Remote: the client does not use the URI concerning matB on *yume* in the input list used for the remote call.
– Local/Local: all URIs are used for in remote call.

This simple experiment aims to show both 1) the seamless multi-protocol management of the library, as well as 2) the possibility for the dispatcher, described page 9, to perform a schedule: due to its priority matching combined with its Round-Robin algorithm, the library uses the local data first if available. Figure 4 clearly shows this behavior, the blue region showing the time spent during each transfer when it occurs.

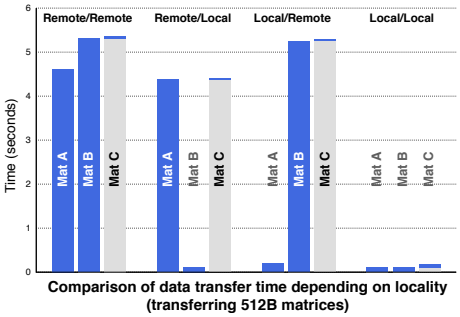

Fig. 4. Results for Experiment 1

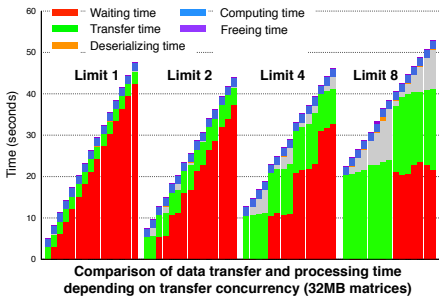

Fig. 5. Results for Experiment 2

4.2 Asynchronous transfers management and bounded number of simultaneous transfers

For this experiment, we designed the following scenario: a remote procedure call is performed to add a given number of matrices which are available remotely through `ssh`. Matrices are downloaded and added as soon as the operation is possible, *i.e.,* at first when two of them are finished to be downloaded, then every time a new one has been downloaded and the previous computation has finished. We performed 4 tests, corresponding to the number of simultaneous transfers that it is possible to make at a given instant, resp. 1 unique transfer, 2, 4 and 8 transfers maximum at a given time. The number of matrices is fixed to 16, and one matrix is 32MB (*i.e.,* 2000×2000 of 64bits integers).

We designed this experiment to validate the request controller behavior, *i.e.,* the implementation of the possibility to limit the number of simultaneous transfers occurring in a transfer operation, and the possibility to use the waiting functions, here with `grpc_data_transfer()` and the `GRPC_WAIT_ANY` parameter, making the server able to perform an operation as soon as enough matrices

are present on the server side (the addition was chosen for the operation since it requires less time than a transfer, which leads to show the wanted behavior. Besides, it also makes sense conceptually since it's a commutative operation).

Figure 5 shows the activity of the service and its duration on the y-axis, related to the progression over the number of matrices downloaded for each limit on the x-axis. The same evolution by group of cardinal equal to the possible limit of both the waiting time (non-active wait) and transfer time highlights that the number of simultaneous transfers operated by the library is indeed configurable (for the moment the information is static in the configuration file. We intend to look if it makes sens to have it self-tuned by the library, depending on the dynamicity of both the network and computing performance). It also shows that every computations occur when enough matrices are finished to be downloaded. As a side effect, it also confirms the observations made in [8]: there's a real need to limit the number of possible parallel transfers. We can indeed observe on this small example that the overall completion time of the addition of the 16 matrices is a bit reduced when the limit is fixed to 2 for our small testbed.

5 Future works

Future works are heading towards different directions. If the library is already usable and implements most of the API, more performance can be obtained with more efficient scheduling: at the request controller (Section 3.2), and at the dispatcher level (Section 3.3); and more development: for example including a middleware module for `ssh` would add more scheduling possibilities; the protocol `memory` leads to already complex data management mechanisms, yet to be continued in addition to a `file` protocol that would help avoiding useless data copies, making the use of the library even more scalable. Modules for dCache and GridFTP would possibly make transfers faster, but further control would have to be done on the bandwidth consumption; a data manager module for Amazon S3[6] would give further access to cloud storage resources leading for a need to also take into account some financial criteria in the above scheduling process, and possible migration of data when possible (*e.g.,* when the data is requested as GRPC_PERSISTENT).

6 Conclusion

With the GridRPC Data Management standard completing previous works on GridRPC, both at the API and software level, feasibility of computations and performance is at reach with immediate portability and interoperability between GridRPC middleware. To ease its spread, while giving access to GridRPC middleware and to existing data managers, we provide an implementation of both APIs relying on a very modular architecture. Fulfilling the standard requirements, the library also implements the data management modes as well as a

[6] `http://aws.amazon.com/`

`memory` protocol to avoid useless copy to disk. We showed that an efficient system to handle waiting mechanisms is in place and that we operate some mapping/scheduling when several transfers are involved in the same data management. We conducted some experiments and obtained results validating the expected behaviors. From now on we will focus on more theoretical work to improve the yet non-trivial mapping/scheduling of transfers involved for a given data, and we are considering to plug a workflow/dataflow analyzing tool to schedule transfers of different data with remote procedure calls altogether. Further developments will also occur, giving more adaptability and choices to the end-user while bringing new issues concerning scheduling possibilities, for example with Cloud storage resources.

Acknowledgment: This work is partially founded by the ÉNS Lyon. The authors want to thank Hidemoto Nakada for the Ninf middleware module.

References

1. Yves Caniou, Eddy Caron, Gaël Le Mahec, and Hidemoto Nakada. Transparent Collaboration of GridRPC Middleware using the OGF Standardized GridRPC Data Management API. In *The International Symposium on Grids and Clouds (ISGC)*, page 12p. Proceedings of Science, February 26 - March 2 2012.
2. Yves Caniou, Eddy Caron, Gaël Le Mahec, and Hidemoto Nakada. Data management API within the GridRPC. In *GFD-R-P.186*, June 2011.
3. B. Del-Fabbro, D. Laiymani, J.M. Nicod, and L. Philippe. DTM: a service for managing data persistency and data replication in network-enabled server environments. *Concurrency and Computation: Practice and Experience*, 19(16):2125–2140, 2007.
4. F. Desprez, E. Caron, and G. Le Mahec. DAGDA: Data Arrangement for the Grid and Distributed Applications. In *AHEMA 2008. International Workshop on Advances in High-Performance E-Science Middleware and Applications. In conjunction with eScience 2008*, pages 680–687, Indianapolis, Indiana, USA, December 2008.
5. Frédéric Camillo, Yves Caniou, Benjamin Depardon, Ronan Guivarch, and Gaël Le Mahec. Improvement of the data management in GridTLSE, a sparse linear algebra expert system. *JCIT: Journal of Convergence Information Technology*, 8(6):562–571, 2013.
6. Adrian Muresan. *Scheduling and deployment of large-scale applications on Cloud platforms*. These, Ecole normale supérieure de lyon - ENS LYON, December 2012.
7. Y. Nakajima, Y. Aida, M. Sato, and O. Tatebe. Performance evaluation of data management layer by data sharing patterns for GridRPC applications. In *LNCS Euro-Par 2008 - Parallel Processing*, volume 5168, pages 554–564, 2008.
8. J. McLaren T. Kosar, A. Hutanu and D. Thain. Coordination of access to large-scale datasets in distributed environments. In A. Shoshani, CRC Press/Taylor D. Rotem, and Francis Books, editors, *Scientific Data Management: Challenges, Existing Technology, and Deployment*, 2009.
9. O. Tatebe, K. Hiraga, and N. Soda. Gfarm grid file system. *New Generation Computing*, 28:257–275, 2010.