PROCEEDINGS OF SCIENCE

Transparent Collaboration of GridRPC Middleware using the OGF Standardized GridRPC Data Management API

Yves CANIOU[∗†]
Université de Lyon, CNRS, ÉNS-Lyon, INRIA, UCBL
E-mail: Yves.Caniou@ens-lyon.fr

Eddy CARON[†]
Université de Lyon, CNRS, ÉNS Lyon, INRIA, UCBL
E-mail: Eddy.Caron@ens-lyon.fr

Gaël Le MAHEC
Université de Picardie Jules Verne
E-mail: Gael.Le.Mahec@u-picardie.fr

Hidemoto NAKADA
National Institute of Advanced Industrial Science
E-mail: hide-nakada@aist.go.jp

This paper presents a basic implementation of the OGF standard GridRPC Data Management API and its integration in two different middleware, respectively DIET and Ninf. We have conducted several experiments, showing the benefits a Grid user can expect 1) in terms of computation feasibility and resource usage compared to the current GridRPC context since useless transfers can be avoided; 2) in terms of reducing the completion time of an application to obtain results the soonest (data and temporary results can be easily kept on the computational and storage servers); 3) in terms of code portability, since we show with these examples that at last the same GridRPC code can be compiled and executed within two different GridRPC middleware which implements the GridRPC data management API; 4) finally we thus obtain middleware interoperability without any explicit glue: we show as a proof of concept that resources dispatched across different administrative domains can be used altogether without the underlying distributed data management but this needs knowledge of the topology and/or computing resources: computational servers of DIET and Ninf transparently collaborate to the same calculus by sharing GridRPC data!

The International Symposium on Grids and Clouds (ISGC) 2012,
Febraury 26 - March 2, 2012
Academia Sinica, Taipei, Taiwan

∗Speaker.

†This work is partially founded by the Technological Development Department (D2T) at INRIA.

© Copyright owned by the author(s) under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike Licence.

PoS(ISGC 2012)028

1. Introduction

The GridRPC API [10] was designed to define Remote Procedure Call over the Grid, widely and successfully used in middleware like DIET [7], NetSolve/GridSolve [16], Ninf [14], OmniRPC [13] and SmartGridSolve [4]. The API concentrated on remote service executions: It standardized synchronous and asynchronous calls and computing tasks management.

In June 2011, the Open Grid Forum standardized the document "Data Management API within the GridRPC" [6] which describes an optional API that extends the GridRPC standard [10] (referenced as GridRPC DM API in this paper). Used in a GridRPC middleware, it provides a minimal set of structure definitions and functions to handle a large set of data operations among which: movements, replications, persistence.

For this paper, French and Japanese co-authors of the GridRPC DM API have implemented the corresponding structures mandatory to the use of the API in their own GridRPC middleware, respectively DIET and Ninf. A first prototype of the API containing the basic functions offering data management through the library has been developed, in addition to a GridRPC layer making possible to write a single GridRPC client able to invoke DIET and Ninf services transparently. With several experiments relying on these early developments, we show that the GridRPC DM API answers already most of the needs presented in [5], namely **code portability** (the exact GridRPC code written with the GridRPC DM API to manage data can be used with any GridRPC middleware), **computation feasibility** (because of the transparent use of remote data, a GridRPC client is not required to have data on its own system to respect the GridRPC paradigm, hence leading to the use of light client machines), **performance** (useless transfers can be avoided with the use of persistence). Moreover, we show that it is even possible for two Grid middleware, managing computing resources across different administrative domains and using the API, to collaborate to the resolution of the same calculus in a completely transparent manner.

2. State of the Art

In the GridRPC paradigm, input and output data are arguments of `grpc_call()` and `grpc_call_async()` and are transferred between a client and the invoked server during steps (4) and (5) of Figure 1. Thus, each Grid middleware managing its own built-in data, **code portability** is impossible, and **performance** can only be manages bypassing the GridRPC model, even for request sequencing [2]. Many other issues arise [5], but they can only be addressed separately: one can store data on distributed file system like GlusterFS[1] or GFarm [15] to deal with automatic replication; OmniRPC introduced omniStorage [11] 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 (JuxMEM, DTM, and Dagda [1, 8, 9]), which focus on both user explicit data management and persistence of data across the resources, with transparent migrations and replications.

[1] `http://www.gluster.org/`

PoS(ISGC 2012)028

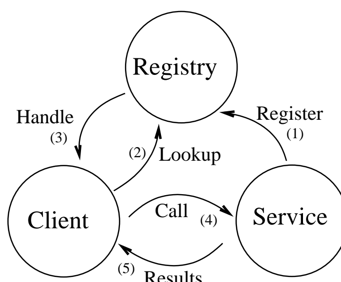

Figure 1: GridRPC paradigm

PoS(ISGC 2012)028

Overall, no solution that can solve all issues can fully rely on existing works: in addition to only fulfilling complementary services, the lack of standard makes their implementation and usage *not interoperable nor portable* through different middleware; thus the need for the GridRPC DM API standards.

3. GridRPC Data Management API

We only present in this section the parts of the GridRPC DM API that have been implemented in our prototype, in terms of data definitions and functions. Full and detailed version is available in [6].

3.1 Data definition

In [10], input/output data parameters are provided within the *variadic arguments* notation of `grpc_call()` and `grpc_call_async()`. The GridRPC DM API uses the `grpc_data_t` as the type of such parameters: it is either a *computational data*, or contains *a reference* on the data (the *Data Handle*) as well as some *Storage Information*. A data handle is essentially a unique reference to a data that may reside anywhere (data and data handles can be created separately). A `grpc_data_t` variable can represent information on a specific data which can be local or remote. It is at least composed of: 1) Two NULL-terminated lists of URIs, one to access and one to record the data (used to prefetch, or transfer it at the end of a computation); 2) The mode of management; 3) Information concerning the type of the data; 4) Data dimensions.
We provide below more details for each item:

1. URI: it defines the location where a data is stored. URI format is described in [3]. It is built using "protocol:[//machine_name][:port]/data_path" and thus contains at least four fields, with protocol being a token like "ibp", "http", "memory" or "middleware" for example. Some fields are optional, depending on the requested protocol.
   For example, the URI ``` ``http://myName/myhome/data/matrix1'' ``` corresponds to the location of a file named `matrix1`, which can be accessed on the machine named `myName`, with the `http` protocol.

2. The management mode is an enumerated type `grpc_data_mode_t` describing the policy values on how to manage data on the platform can be set by the client. If the middleware does not handle the given behavior, it throws an error.

- GRPC_VOLATILE: the user explicitly manages the GridRPC data (location, contained data). The data may not be kept inside the platform after a computation, and can be seen as the default policy of the GridRPC API.
- GRPC_STRICTLY_VOLATILE: used when the data *must not be kept* inside the platform after a computation, for security reasons for example.
- GRPC_STICKY: used when a data is kept inside the platform but cannot be moved between the servers. This is used if the client wants data to be available on the same servers for a second computation for example.
- GRPC_UNIQUE_STICKY: like GRPC_STICKY, but the data cannot be replicated.
- GRPC_PERSISTENT: used when a data has to be kept inside the platform. The underlying data middleware is explicitly asked to handle the data: the data can migrate or be replicated between servers depending on scheduling decisions, and potential coherency issues may arise if the user attempt to modify the data on his own.
- GRPC_END_LIST: this is a simple marker to terminate `grpc_data_mode_t` lists.

3. The type of the data is an enumerated type `grpc_data_type_t` and set by the client. It can take a value among GRPC_BOOL for a boolean value, GRPC_INT for an integer value, GRPC_DOUBLE for a double precision floating point value, GRPC_COMPLEX for a complex number, GRPC_STRING for a character string or GRPC_FILE for a regular file.
4. The dimension is a vector terminated by a zero value, containing the dimensions of the data. For example the matrix $[n\ m\ p\ 0]$ describes a $n \times m \times p$ 3D-matrix.

3.2 Function prototypes

For each GridRPC data, a GridRPC DM middleware uses internally a unique Data Handle as well as additional information about the data, like locations and dimensions. The user provides such information with a call to the **init** function, while the data handle is set to the data it identifies. *Explicit* data exchanges are done using the asynchronous **transfer** function. A GridRPC data can also be **inspected** to get information about the status of the data, its locations, etc. Functions are also given to **wait** after the completion of some transfers. One can **unbind** the handle and the data, and **free** the GridRPC data. Finally, to provide identification of long lived data, data handles can be **saved** and **restored**.

```
grpc_error_t
grpc_data_init(grpc_data_t * data, const char** input_URIs, const char** output_URIs,
               const grpc_data_type_t data_type, const size_t* data_dimensions,
               const grpc_data_mode_t* data_modes);
grpc_error_t
grpc_data_free(grpc_data_t* data, const char** URI_locations);
grpc_error_t
grpc_data_memory_mapping_set(const char* key,
                             void* data );
grpc_error_t
grpc_data_memory_mapping_get(const char* key,
                             void** data );
```


PoS(ISGC 2012)028

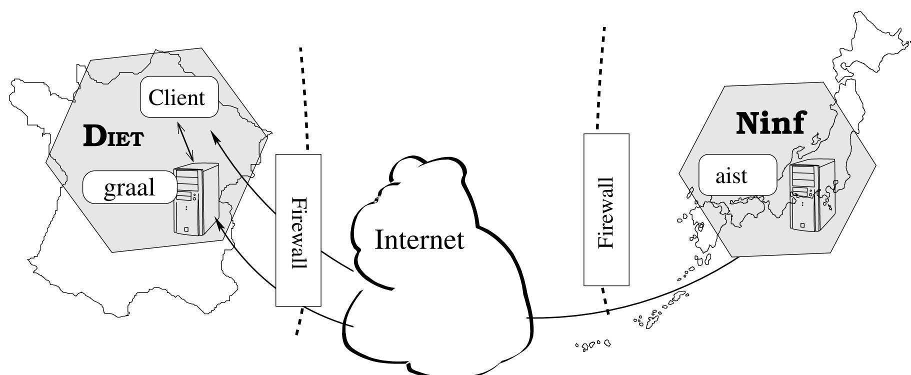

Figure 2: Platform of experimentation

For the present work four functions were required (see page 4). The first two are mandatory to initialize and free a GridRPC data. The two others are needed to manipulate data in RAM, accessible through the `memory` protocol in URIs: they lead to put a relation between the memory location and the string used in URIs that the GridRPC DM library is able to understand to access data.

4. Platform description

This section describes the testbed, depicted in Figure 2, used to realize the different experiments leading to the results given in the next section.

4.1 Resources description

Computing and storage resources involved in our testbed are composed of the following three **heterogeneous** computing machines:

- `aist` is a 8 2.4GHz cpu virtual machine with 22GB RAM running the 3.0.0-15 32bits linux kernel, and located in Tsukuba (Japan) on the SINET network. Used as a GridRPC server.
- `graal.ens-lyon.fr` is a 16 4 core cpu 2.93GHz machine (Intel Xeon X5570), with 32GB RAM and running the 2.6.18-27 64bits linux kernel, located in Lyon (middle east of France) on the RENATER network. Used as a GridRPC server and as a HTTP server,
- `client` is a MacBook Pro core 2 duo 2.4GHz with 4GB RAM, running MacOS 10.6.8 64bits, located in Amiens (north of France) on an ADSL 2+ connection. Used as GridRPC client.

4.2 Connectivity measurements

We give in Table 1 the mean bandwidth between all equipment on the testbed. The available bandwidth is calculated with the use of `scp` (due to firewall constraint) by timing 24 transfers of 32 MB matrices between `graal` and `aist`, and 2MB matrices between `client` and the two others.

PoS(ISGC 2012)028

For the same reason (firewalls), we could not easily measure the latency. From the simple `traceroute`, we know that it should be a bit more than 300msec. Note that a connection between `graal` and `aist`, *i.e.,* between RENATER and SINET, goes through the GEANT2 network, and such a connection has been measured to be around 10Gbs bandwidth with a RTT of about 290msec in 2009 for studies on LHC-2, which is still correct according to our tests.

↱	aist	graal	client
aist	-	8271.20	1013.73
graal	11260.01	-	2454.07
client	569.52	739.23	-

Table 1: Bandwidth across the platform, in Kbits/s

With the help of those values, we could compare the results obtained in the next section to computed "expectation times".

5. Results

5.1 Stickiness and remote data

We designed two scenarios to highlight the most obvious benefits of using the GridRPC DM API implementation library, in terms of reducing the resources cost (less resource used, no waste) and the computation time of the execution of an application. For each scenario we conducted 3 different experiments (variants, noted S_iE_j for Scenario i Experiment j) to compare observed timings of the scenario in the standard GridRPC context to the GridRPC DM context with the help of very simple use case by 1) using stickiness and 2) also considering remote data.

Scenario 1 is the sum of n matrices where each matrix A_i is a 1000×1000 matrix and only the service `matsum`, performing the sum of two matrices, is available on the platform.

Figure 3 shows S_1E_1, the most simple case in a GridRPC context. We assume that data are available on the client side, and the service is called n times, each time involving 3 transfers: both a transfer of two matrices during the `grpc_call()` and when the resulting matrix is sent back. Results given in Figure 5 shows the timings for different number of n. As one can see, the total duration of a run is leaded by the time to transfer data. One can see that in this example, the computation time is negligible. On can argue that it would have been better to perform the computations locally, but, if this is just a use-case example, in a real case prior additional transfers to download matrices on the client would also have been to consider.

Figure 6 shows the benefits of the possibility to keep temporary results on the computational server. Depicted in Figure 4, for $n > 2$ we spare $2 \times (n-2)$ matrix transfers, leading to a linear improvement, *e.g.,* a ratio of 5.3 on the run duration for $n = 10$ and a bandwidth ∼2.5Mbits/s.

At last we present in Figure 7 the timings when a client uses both the remote data management and the stickiness: data are available on a HTTP server and are transferred from there at each call; temporary results are kept on the computational server (Figure 8). The result is sent to the HTTP server instead of the client. In our testbed the computational server is also the HTTP server, making HTTP downloads local and the results the best as it could be.

PoS(ISGC 2012)028

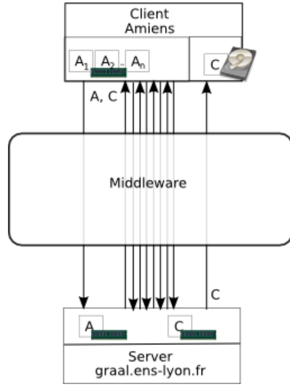

Figure 3: S_1E_1: no sticky

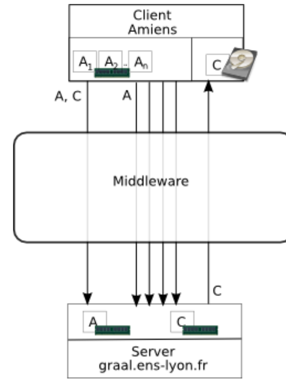

Figure 4: S_1E_2: sticky

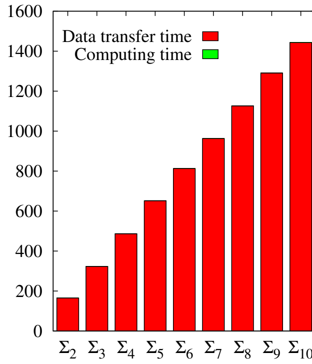

Figure 5: S_1E_1: no sticky

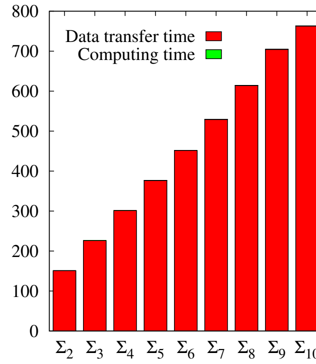

Figure 6: S_1E_2: sticky

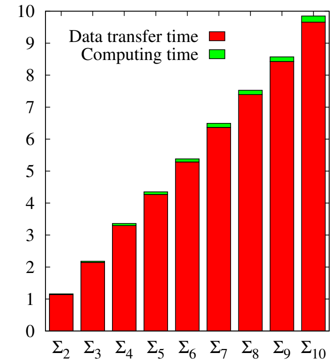

Figure 7: S_1E_3: HTTP data

Scenario 2 is the computation of A^n where A is a 1000×1000 matrix and only the service `matprod`, performing the product of two matrices, is available on the platform.

The most simple GridRPC case for this scenario can also be depicted in Figure 3 in terms of transfers, with all A_i equals to A. We consider indeed that A is already available on the client, and the service is called n times, each time involving 3 transfers, like for S_1E_1. Results given in Figure 10 show that a run duration is mainly leaded by involved transfers.

Using the stickiness for temporary results leads to measurements reported in Figure 11. One can see that the transfer duration is quiet constant regardless n: indeed as pictured in Figure 9, there is only two transfers involved in a run, one to upload A on the computational server onto which it is kept sticky (as are temporary results), and the transfer of the result. `grpc_call()` are made using the updated `grpc_data_t` containing the URIs of the location of data (*i.e.,* on the server), and only prior to the final call the output is updated so that the resulting matrix is sent to the client.

In case of A is available from the HTTP server and we want the result available there as well, we can use the remote data management and the stickiness, as we did for S_1E_3 except that in this case, (Figure 13), only one transfer for input data is needed, A being kept sticky during the computation. Only two transfers are involved as well, but since data can be considered as local to the GridRPC server, we obtain the best possible performance.

PoS(ISGC 2012)028

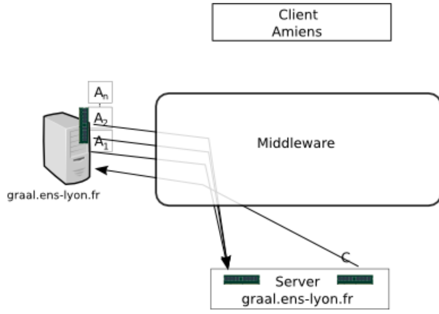

Figure 8: S_1E_3: HTTP data

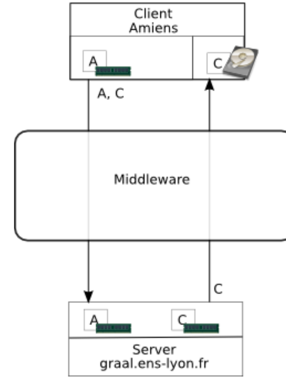

Figure 9: S_2E_2: sticky

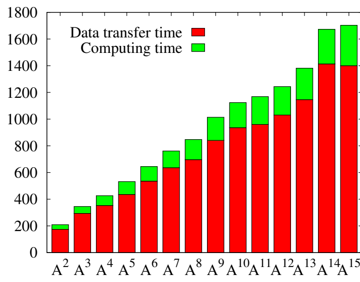

Figure 10: S_2E_1, no sticky

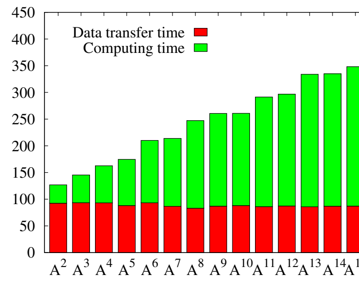

Figure 11: S_2E_2, sticky

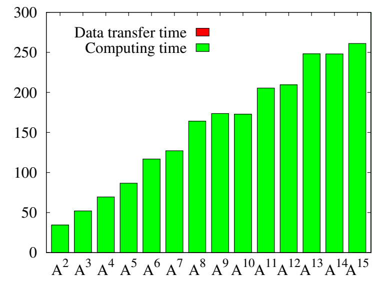

Figure 12: S_2E_3, HTTP data

5.2 Inter-middleware transparent collaboration

Consider the situation where two GridRPC servers are available: the first one is the DIET GridRPC server `graal` that provides the service `matsquare` which performs the square of an input matrix; the second one is the Ninf GridRPC server `aist` providing the service `matprod`, performing the product of two matrices. A GridRPC user wants to perform the computation $(A \times B)^3$ for two given matrices A and B. Considering the available computing resources, the user can perform $A \times B \times A \times B \times A \times B$ using exclusively `aist` (normal GridRPC mode since stickiness is not yet implemented in Ninf), or use a workflow (Figure 14) and compute $C = A \times B$ on `aist` (step (1)), then $D = C^2$ on `graal` (step (2)), and $E = C \times D$ on `aist` (step (3)) for example.

In a general context, using the workflow would be a huge task to perform: 1) on a practical point of view, it involves two GridRPC middleware that have their own independent libraries. The fact that they provide their own GridRPC API layer on top of their mnemonics also makes it harder to deal with, and one way would be to realize a GridRPC client that calls in forked processes the corresponding `grpc_call()` in external binaries. 2) the client has to take into account the monitoring of the availability of intermediary data, when they are produced and transferred, as well as the interoperability between the middleware frameworks, since they manage and store data their own way, etc. 3) Besides in the GridRPC model, data are available on the client side when performing the calls, which may introduce additional delays to the overall computation duration, and a great care on the ratio realization/maintenance and gain has to be studied. Overall, chances that such a solution would ever be considered is low. Nevertheless, we implemented a fully work-

PoS(ISGC 2012)028

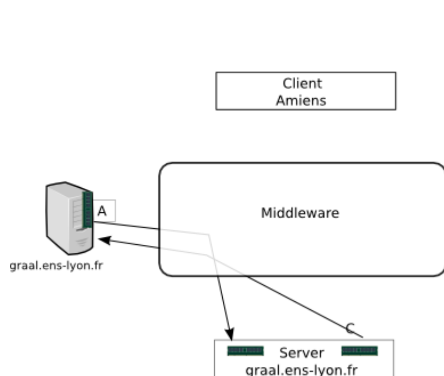

Figure 13: S_2E_3: HTTP data

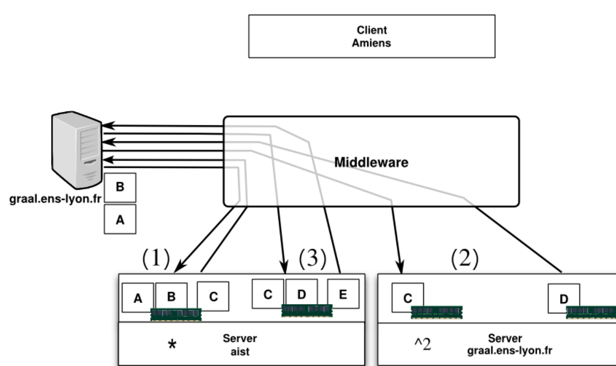

Figure 14: Schema of DIET-Ninf collaboration

Experiment	Duration (1)	Duration (2)	Duration (3)
Ninf only	95724	169513	159266
Collaboration	95780	2966	174570

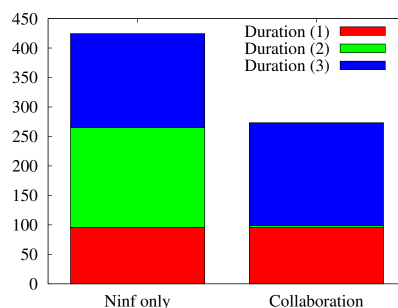

Figure 15: GridRPC collaboration, time in msec (table), in sec (graphe)

ing client which performs this computation, mainly relying on the GridRPC DM API: GridRPC middleware framework are not even aware of interacting to solve a unique workflow.

We have conducted two experiments, and we present in Figure 15 the average recorded times performed on 10 runs of each experiment. As one can see, using the fast computational server managed by the GridRPC DIET server, `graal`, leads to nearly halve the overall computation since step (2) is 57 times faster than on `aist`, managed by Ninf, making the collaboration of the two of them very useful for a Grid user.

5.3 Implementation details and additional remarks

- About managed transfer protocols:
  - HTTP protocol management: to upload the data we used a HTTP POST form method. The main advantage of this solution is that PHP is a widespread script language: it is very easy to set-up and test our solution. Moreover, the script can easily be extended to manage other API parts, like `grpc_getinfo()`. In our library prototype, HTTP transfers are performed using the libCURL open source library.
  - DAGDA protocol management: thanks to the DIET *Forwarder* component, we did not need to modify the way data are transferred to bypass firewalls: we used the DIET's data manager DAGDA functionalities called from the GridRPC DM API implementation.

PoS(ISGC 2012)028

- `memory` is a pseudo protocol used to manage data stored in memory (see Section 3.2). The transfer of such data is performed using one of the available protocols allowing to download/upload the data to/from the service execution node. In our implementation, memory data are simply associated to a name using a STL map<string, void*>.

- About service calls: the client prototype needs to manage two different GridRPC middleware frameworks, resp. DIET and Ninf, at the same time. The two parts of the implementation are very different but illustrate that it is possible to use the API in spite of hard technical constraints. Indeed all the computing nodes are installed on servers behind firewalls.
  - Within DIET, using the *Forwarder* component, the parameters of the GridRPC call function are simply converted to the corresponding DIET structures. The service takes three strings for the two input matrices and the output one. These strings store URIs to download the data and to upload the data after the computation. Then everything is managed by the service itself using the same implementation for the HTTP protocol than the client and the DAGDA classical API.
  - Ninf natively supports GridRPC Data structures and the data itself is transmitted to the server via the Ninf transport layer. The server then retrieves the data using HTTP and data mangling modules.

For the experiments we used three different systems/architectures: The client was launched on a MacOS X 64bits systems, the Ninf computing node on a virtualized 32bits Linux system and the DIET node on a 64 bits SMP Linux system. The main difficulty was to manage matrix serialization/deserialization on different architectures. The matrices data format was not affected by the architecture because their storage was conforming to the double-precision IEEE 754 standard but all the attendant meta-data like matrix sizes, storage order, needed a conversion to be shared between the different architectures. For a complete implementation, it should be preferable to use a standard data format description such as the OGF DFDL language [12].

Such details will be integrated in the OGF GridRPC working group “Interoperable document” which aims to describe implementation recommendations so that any implementation of the GridRPC DM API, and thus anything concerning its usage, is fully interoperable with others.

6. Conclusion and future work

We have presented the first evaluation of the benefits leaded by the extension of the GridRPC API OGF standard concerning data management, *i.e.,* the GridRPC Data Management GFD-R-P.186 OGF standard. We have conducted several experiments with its implementation prototype, thus showing the expectations that a user can have in terms of: code portability across different GridRPC middleware frameworks (codes can fully compile with DIET or Ninf); computation feasibility, with the integration of the management of remote data directly taken into account into GridRPC call (synchronous and asynchronous); improved performances (less resource waste, applications may complete sooner) and transparent collaboration to a calculus using different GridRPC middleware frameworks in different administrative domains on heterogeneous architectures.

PoS(ISGC 2012)028

Future work will go towards both experimental material with the development of the rest of the API, also taking into account other protocols such as `GridFTP`, and continue the standardization work with an interoperable document describing the definition of some output GridRPC DM functions like `getinfo()` so that a code can still be written for every GridRPC framework.In particular, this document will describe how to define and interpret the metadata. The standard was designed to facilitate the implementation of different data transfer protocols like FTP, GridFTP, bittorrent, etc. but without making such implementation compulsory for the GridRPC middleware to be standard compliant. We plan to propose recommendations and implementation examples based on the experience we have gained developing the prototype presented in this paper.

Finally, the recommendations, future implementations and possible extensions to the API will depend on the final users/developers remarks. We are eager to get different use-cases and users feedback.

References

[1] G. Antoniu, M. Bertier, L. Bougé, E. Caron, F. Desprez, Mathieu Jan, S. Monnet, and P. Sens. GDS: An architecture proposal for a grid data-sharing service. In V. Getov, D. Laforenza, and A. Reinefeld, editors, Future Generation Grids, volume XVIII, CoreGrid Series of Proceedings of the Workshop on Future Generation Grids November 1-5, 2004, Dagstuhl, Germany. Springer Verlag, 2006.

[2] D.C. Arnold, D. Bachmann, and J. Dongarra. Request Sequencing: Optimizing Communication for the Grid. In EUROPAR: Parallel Processing, 6th International EURO-PAR Conference. LNCS, 2000.

[3] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 3986: Uniform Resource Identifier (URI): Generic syntax. The Internet Society, 2005.

[4] T. Brady, M. Guidolin, and A. Lastovetsky. Experiments with smartgridsolve: Achieving higher performance by improving the GridRPC model. In The 9th IEEE/ACM ICGC, 2008.

[5] Y. Caniou, E. Caron, G. Le Mahec, and H. Nakada. Standardized Data Management in GridRPC Environments. In 6th International Conference on Computer Sciences and Convergence Information Technology, ICCIT'11, Jeju Island, Korea, Nov. 29 - Dec. 1 2011. IEEE.

[6] Y. Caniou, E. Caron, G. Le Mahec, and H. Nakada. Data Management API within the GridRPC. In GFD-R-P.186, June 2011.

[7] E. Caron and F. Desprez. DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid. In International Journal of High Performance Computing Applications, volume 20(3), pages 335–352, 2006.

[8] 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.

[9] 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.

[10] H. Nakada, S. Matsuoka, K. Seymour, J.J. Dongarra, C. Lee, and H. Casanova. A GridRPC Model and API for End-User Applications. In GFD-R.052, GridRPC Working Group, June 2007.

PoS(ISGC 2012)028

[11] 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.

[12] A.W. Powell, M.J. Beckerle, and S.M. Hanson. Data format description language (DFDL) v1.0 specification. In GFD-R-P.174, January 2011.

[13] M Sato, M Hirano, Y Tanaka, and S Sekiguchi. Omnirpc: a GridRPC facility for cluster and global computing in openMP. OpenMP Shared Memory Parallel Programming, 2104:130–136, 2001.

[14] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. Journal of Grid Computing, 1(1):41–51, 2003.

[15] O. Tatebe, K. Hiraga, and N. Soda. Gfarm grid file system. New Generation Computing, 28:257–275, 2010.

[16] A. YarKhan, J. Dongarra, and K. Seymour. GridSolve: The Evolution of Network Enabled Solver. In James C. T. Pool Patrick Gaffney, editor, Grid-Based Problem Solving Environments: IFIP TC2/WG 2.5 Working Conference on Grid-Based Problem Solving Environments (Prescott, AZ, July 2006), pages 215–226. Springer, 2007.

7. Appendix

We give here as an example the C++ code of a client requesting the product of a 100×100 double matrix A available in `memory` by a 100×100 double matrix $data2$ available on the HTTP server `data.test2.fr`, the resulting matrix being sent to the HTTP server running on the client.

```
#include "gridrpc.hh"
#include <iostream>
#include <cstring>

int main(int argc, char* argv[]) {
  grpc::grpc_data_t d1, d2, d3;

  grpc::grpc_function_handle_t handle;

  grpc::grpc_function_handle_default(&handle, "MATMul");

  grpc::grpc_data_init(&d1, NULL, NULL, grpc::GRPC_DOUBLE, NULL, NULL);
  grpc::grpc_data_init(&d2, NULL, NULL, grpc::GRPC_DOUBLE, NULL, NULL);
  grpc::grpc_data_init(&d3, NULL, NULL, grpc::GRPC_DOUBLE, NULL, NULL);

  double* mat = (typeof mat) calloc(100*100, sizeof *mat);

  grpc::grpc_data_memory_mapping_set("a", mat);

  d1 << grpc::IN << "memory://localhost/a";
  d1 << (size_t) 100 << (size_t) 100;
  d1 << grpc::GRPC_VOLATILE;

  d2 << grpc::IN << "http://data.test2.fr/data2";
  d2 << grpc::OUT << "http://data.test2.fr/data2" << "ftp://test2";
  d2 << (size_t) 100 << (size_t) 100;
  d2 << grpc::GRPC_VOLATILE;

  d3 << grpc::OUT << "http://localhost/c";
  d3 << (size_t) 100 << (size_t) 100;
  d3 << grpc::GRPC_VOLATILE;

  try {
    grpc_call_data(&handle, &d1, &d2, &d3);
  } catch (const char* err) {
    std::cerr << "Error: " << err << std::endl;
  }

}
```

A GridRPC code example

PoS(ISGC 2012)028