Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

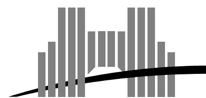
Towards a Transparent Data Access Model for the GridRPC Paradigm

Gabriel Antoniu ,
Eddy Caron ,
Frédéric Desprez ,
Mathieu Jan

October 2006

Research Report N° 2006-47

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`



Towards a Transparent Data Access Model for the GridRPC Paradigm

Gabriel Antoniu , Eddy Caron , Frédéric Desprez , Mathieu Jan

October 2006

Abstract

As grids become more and more attractive for solving complex problems with high computational and storage requirements, the need for adequate grid programming models is considerable. To this purpose, the GridRPC model has been proposed as a grid version of the classical RPC paradigm, with the goal to build NES (Network-Enabled Server) environments. Paradoxically enough, in this model, data management has not been defined and is now explicitly left at the user's charge. The contribution of this paper is to enhance data management in NES by introducing a *transparent data access model*, available through the concept of grid data-sharing service. Data management (persistent storage, transfer, consistent replication) is totally delegated to the service, whereas the applications simply access shared data via global identifiers. We illustrate our approach using the DIET GridRPC middleware and the JUXMEM data-sharing service. Experiments performed on the Grid'5000 testbed demonstrate the benefits of the proposed approach.

Keywords: GridRPC, Data Sharing, Persistency, JUXMEM, DIET.

Résumé

À mesure que les grilles deviennent de plus en plus attractives pour résoudre des problèmes complexes nécessitant d'importantes capacités de calcul et de stockage, le besoin de modèles de programmation adéquats pour ces architectures grandit. Dans ce but, le modèle GridRPC a été proposé comme une version pour les grilles du paradigme de programmation par RPC, avec pour objectif de construire des plate-formes logicielles de calcul. Paradoxalement, dans ce modèle, la gestion des données n'a pas été définie et est pour l'instant explicitement laissée à la charge de l'utilisateur. La contribution de ce papier est d'améliorer la gestion des données dans ces plate-formes logicielles de calcul en introduisant un *modèle d'accès transparent aux données*, disponible par l'utilisation du concept de service de partage de données pour grilles. La gestion des données (stockage persistant, transfert, réplication cohérente) est totalement déléguée à ce service, tandis que les applications accèdent aux données partagées via des identifiants globaux. Nous illustrons notre approche en utilisant l'intergiciel GridRPC DIET et le service de partage de données pour grilles JUXMEM. Les expériences réalisées sur la grille expérimentale Grid'5000 démontre les bénéfices de l'approche proposée.

Mots-clés: GridRPC, Partage de données, Persistance, JUXMEM, DIET.

1 Introduction

Computational grids have recently become increasingly attractive, as they adequately address the growing demand for resources of today's scientific applications. Thanks to the fast growth of high-bandwidth wide-area networks, grids efficiently aggregate various heterogeneous resources (processors, storage devices, network links, etc.) belonging to distinct organizations. This increasing computing power, available from multiple geographically distributed sites, increases the grid's usefulness in efficiently solving complex problems.

An example of class of applications that can benefit from grid computing consists of multi-parametric problems, in which the same algorithm is applied to slightly different input(s) data in order to obtain the best solution. Running such applications on grid infrastructures requires the use of adequate programming paradigms for distributed systems, able to cope with the targeted large scale. Among the various programming models which aim at reducing the increasing programming complexity, one approach relies on the *Grid Remote Procedure Call* (GridRPC) [19]. This paradigm extends the classical RPC model by enabling asynchronous, coarse-grained parallel tasking. GridRPC seems to be a good approach to build NES computing environments (for Network-Enabled Servers). In such systems, clients can submit problems to one (possibly distributed) agent, which selects the best server among a large set of candidates.

A team of researchers of the Global Grid Forum (GGF, recently merged with EGA (Enterprise Grid Alliance) to create the OGF (Open Grid Forum)) has defined a standard API for the GridRPC paradigm [16]. However, in this specification, data management has been left as one open (although fundamental) issue. For instance, data transfer in the distributed environment is left to the user: data have to be explicitly moved back and forth from the clients to the selected servers. This is clearly a major limitation for efficiently programming grids, especially as the number of servers used to solve a problem increases.

In this paper, we define a model for transparent access to shared data in GridRPC environments. In this model, the data-sharing infrastructure automatically manages data localization, transfer, as well as consistent data replication. We illustrate our approach with an implementation using the DIET [10] GridRPC middleware and the JUXMEM [3] grid data-sharing service. We evaluate our approach through experiments realized on the Grid'5000 [8] testbed.

The remainder of the paper is organized as follows. Section 2 introduces the GridRPC model, a motivating application for data management and briefly describes previous attempts to solve data management issues in NES systems. Section 3 describes our transparent data access approach provided by our concept of grid data-sharing service. Section 4 presents the implementation of our proposal, using JUXMEM and DIET. Section 5 presents and discusses our experimental results. Finally, Section 6 concludes the paper and suggests possible directions for additional research.

2 Data management in the GridRPC model

Various programming models have been proposed in order to reduce the programming complexity of grid applications. The GridRPC model is such an ongoing work carried out by the Global Grid Forum (GGF), with the goal of standardizing and implementing the Remote Procedure Call (RPC) programming model for grid computing.

2.1 The GridRPC model

The GridRPC model enhances the classical RPC programming model with the ability to invoke asynchronous, coarse-grained parallel tasks. Requests for remote computations may indeed generate parallel processing, however this server-level parallelism remains hidden to the client.

The GridRPC approach has been defined in the GRIDRPC-WG [24] working group of the GGF. The goal of this group is to specify the syntax and the programming interface at the client level [19]. This is meant to enhance the portability of GridRPC applications to various GridRPC middleware.

The GridRPC model aims at serving as a basis for software infrastructures called Network-Enabled Servers (NES). Such infrastructures allow the concurrent execution of multiple applications on a shared

set of grid resources. Example of middleware that implement the GridRPC specification are Ninf-G [20], NetSolve [5], GridSolve [22], DIET [10], and OmniRPC [18].

In the GridRPC model, *servers* register *services* to a *directory*. To invoke a service, *clients* first look for a suitable (if not "the best", according to some performance metric) server among a large set of candidates proposed by the directory. Note that the GridRPC model does not define any standard for the mechanism used for resource discovery. The choice of a server is performed by one or several *agents*, which implement the directory functionnality of the GridRPC model. The decision is usually based on performance information, provided by an information service. Informations can be static, such as processor speed or size of the memory, but also dynamic, such as installed services, the load of the server as well as the location of input data, etc. Based on this, the goal of the agents is to optimize the overall throughput of the platform.

Two fundamental concepts in the GridRPC model are the *function handle* and the *session ID*. The function handle represents a binding between a service name and an instance of that service available on a given server. Function handles are returned by agents to clients. Once a particular function-to-server mapping has been established, all GridRPC calls of a client will be executed on the server specified by that function handle. A session ID is associated to each asynchronous GridRPC call and allows to retrieve the status of the request, wait for the call to complete, etc. Based on these two concepts, the interface of the GridRPC model mainly consists of the following two functions: `grpc_call` and `grpc_async`, which allow to make synchronous and asynchronous GridRPC calls respectively.

As regards data, most GridRPC middleware systems specify three *access modes* (also known as *access specifiers*) for parameters of a GridRPC call: 1) `in` data for input parameters that are not allowed to be modified by servers; 2)`inout` data for input parameters that can be modified by the server; 3) `out` data for output parameters produced by the server.

2.2 Requirements for data management in the GridRPC model

To illustrate the requirements related to data management in the GridRPC model, we have selected the Grid-TLSE project [11]. This application aims at designing a Web portal exposing expertise about sparse matrix manipulation. Through this portal, the user may gather actual statistics from runs of various sophisticated sparse matrix algorithms on his/her specific data. The input data are either submitted by the user, or picked up from the matrix collection available on the site. In general, matrix sizes can vary from a few megabytes to hundreds of megabytes. The Grid-TLSE application uses the DIET GridRPC middleware to distribute tasks over the underlying grid infrastructure. Each such task consists in executing a parallel solver, such as MUMPS [1], over a matrix, with fixed parameters.

A typical scenario consists in determining the *ordering sensitivity* of a class of solvers, that is, how performance is impacted by the matrix traversal order. It consists of three phases. Phase 1 exercises all possible internal orderings in turn. Phase 2 computes a suitable metric reflecting the performance parameters under study for each run: effective FLOPS, effective memory usage, overall computation time, etc. Phase 3 collects the metric for all combinations of solvers/orderings and reports the final ranking to the user. If phase 1 requires exercising n different kinds of orders with m different kinds of solvers, then $m \times n$ executions are to be performed, using the same input data. If the server does not provide *persistent storage*, the matrix has to be sent $m \times n$ times to the server! If the server provided persistent storage, the data would be sent only once. Second, if the various pairs solvers/orderings are handled by different servers in phase 2 and 3, then *transparent* and *consistent* data transfers or replication across servers should be provided by the data management service. Finally, as the number of solvers/orderings is potentially large, many nodes are used. This increases the probability for faults to occur, which makes the use of *fault tolerant* algorithms to manage data mandatory.

Based on this example, we can draw the requirements for a data management service for the GridRPC model.

Persistent storage. Clients should be able to invoke services on input data that is already present on the grid infrastructure, to avoid repeated data transfers to servers.

Argument passing by reference for shared data. This is a consequence of the above requirement, as clients need a means to reference data which is shared by multiple GridRPC calls. This implies the need to also ensure data consistency in case of concurrent accesses.

Transparent data localization and transfer. This would allow GridRPC applications to scale, as developpers need no longer manage these aspects explicitly.

Efficient communication. An efficient use of the available bandwidth for data transfers means to adequatly manage data granularity: only the subsets of data that are needed to perform computations should be copied or moved.

GridRPC interoperability. Any solution addressing the above issues needs to be compatible with the existing core API of the GridRPC model. Thus current applications can take advantage of any improvement in data management without modifications.

2.3 Current proposals for data management in the GridRPC model

In the current GridRPC model, as defined by the GGF, data persistence is not provided and has been left as an open issue. Therefore, output data of a computation (`inout` and `out`) are systematically sent to the client, whereas input data (`in`) are destroyed on the server. Hence, data needs to be transfered again if needed for another computation. Moreover, if data are required on multiple servers at the same time, multiple transfers from the client are needed.

The issue of data management in the GridRPC model has however been recognised as a topic of major interest. The very first proposal related to data management relies on the concept of *request sequencing* [6]. This feature consists in scheduling a sequence of GridRPC calls made by a client on a given server. In the client program, a sequence is identified by keywords `begin_sequence` and `end_sequence`. Data movements due to dependencies in calls between such keywords are then optimized. Request sequencing has been implemented in NetSolve and Ninf. To enable the calls of a sequence to be solved in parallel on two different servers, NetSolve has been enhanced [12] with data redistribution between servers (which however requires explicit calls in the NetSolve client application).

Another approach for data management relies on distributed storage infrastructure, such as Internet Backplane Protocol (IBP [7]). In this approach, clients send data to storage servers, which retrieve data as needed. NetSolve has been modified in such a way. However, data is still explicitly transfered to/from the storage servers at the application level. Besides, no support for data replication and consistency management, nor for fault tolerance is provided.

Finally, other GridRPC systems have developped ad-hoc, specific mechanisms for data management. The OmniRPC GridRPC middleware supports a static persistence model for input data of a set of GridRPC calls [17]. The user has to manually define a initialization procedure to indicate which input data should be sent and store in advance to executions. Then, these data can be reused for subsequent calls. In an earlier version, the DIET GridRPC middleware relies on an internal data management system, called Data Tree Manager (DTM), which allows to store persistent data [14] on the computing servers. However, as in both cases ad-hoc solutions are used to handle data persistence, this goes against GridRPC interoperability, as data cannot be shared among multiple GridRPC middleware frameworks. Besides, none of these solutions addresses fault tolerance and consistent replication.

Based on such preliminary efforts, an attempt to standardize data management in NES is currently being pursued in the framework of the GridRPC working group of the GGF [16]. It relies on the concept of *data handle*, which abstracts a given data as well as its location. In addition to the possibility of referencing data stored inside external storage systems via a binding mechanism, transparent access to data is also envisioned. However, replication, consistency guarantees and fault tolerance issues are still not addressed.

3 Our approach: a transparent data access model

3.1 The concept of data-sharing service

Let us recall that one of the major goals of the grid concept is to provide an easy access to the underlying resources, in a *transparent* way. The user should not need to be aware of the localization of the resources allocated to applications. When applied to the management of the data used and produced by applications, this principle means that the grid infrastructure should automatically handle data storage and data transfer

among clients, computing servers and storage servers as needed. It should also handle fault tolerance and data consistency guarantees in such dynamic, large-scale, distributed environments and again, in a transparent way.

In order to achieve a real virtualization of the management of large-scale distributed data, a step forward has been made by proposing a *transparent data access model*, as a part of the concept of *grid data-sharing service* [2]. This concept of grid data-sharing service has been illustrated by the JUXMEM software experimental platform [3]. In this *transparent data access* approach, the user accesses data via global identifiers, which allow to do argument passing by reference for shared data. The service which implements this model handles data localization and transfer without any help from the programmer. The data sharing service concept is based on a hybrid approach inspired by DSM systems (for transparent access to data and consistency management) and peer-to-peer (P2P) systems (for their scalability and volatility tolerance). The service specification includes three main properties.

Persistence. The data sharing service provides persistent data storage and allow the applications to reuse previously produced data, by avoiding repeated data transfers between clients and servers.

Data consistency. Data can be read, but also *updated* by the different codes. When data is replicated on multiple sites, the service has to ensure the consistency of the different replicas, thanks to *consistency models and protocols*.

Fault tolerance. The service has to keep data available despite disconnections and failures, e.g. through replication techniques and failure detection mechanisms.

Let us note that these properties match well the requirements for data management in the GridRPC model, as discussed in Section 2.2. We therefore propose to jointly use the two approaches. In this paper we mainly focus on persistence. A full description of concepts and technical details related to data consistency and fault tolerance (equally important for NES) is beyond the scope of this paper and has been detailed in [4].

3.2 Overview of the JUXMEM data-sharing service

The concept of data-sharing service has been illustrated by the JUXMEM [3] software experimental platform. The architecture of the service has been designed so as to address the properties mentioned in Section 3.1. JUXMEM's architecture mirrors a grid consisting of a federation of distributed clusters and is therefore *hierarchical*. The goal is to accurately map the physical network topology, in order to efficiently use the underlying high performance networks available on grid infrastructures. Consequently, the architecture of JUXMEM relies on node sets to express the hierarchical nature of the targeted testbed. They are called *cluster groups* and correspond to physical clusters (see Figure 1). These groups are included in a wider group, the *juxmem group*, which gathers all the nodes running the data-sharing service.

Any cluster group consists of *provider* nodes which supply memory for data storage. Each cluster group is managed by special peer, called a *manager*. These managers are making up the backbone of a given JUXMEM network, and handle the propagation of memory allocation requests. Any node (including providers) may use the service to allocate, read or write data as *clients*, in a peer-to-peer approach. Any data stored in JUXMEM is transparently accessed through a global, location-independent identifier, which designates a specific *data* group that includes all replicas of that data. These replicas are kept consistent despite possible failures and disconnections [4]. This software architecture has been implemented using the JXTA [21] generic P2P platform.

3.3 JUXMEM from the user's perspective

The programming interface proposed by the the JUXMEM grid data-sharing service provides users with classical functions to allocate and map/unmap memory blocks, such as `juxmem_malloc`, `juxmem_calloc`, etc. When allocating a memory block, the client has to specify: 1) on how many clusters the data should be replicated; 2) on how many providers in each cluster the data should be replicated; 3) the consistency protocol that should be used to manage this data. The allocation operation returns a global data ID. This

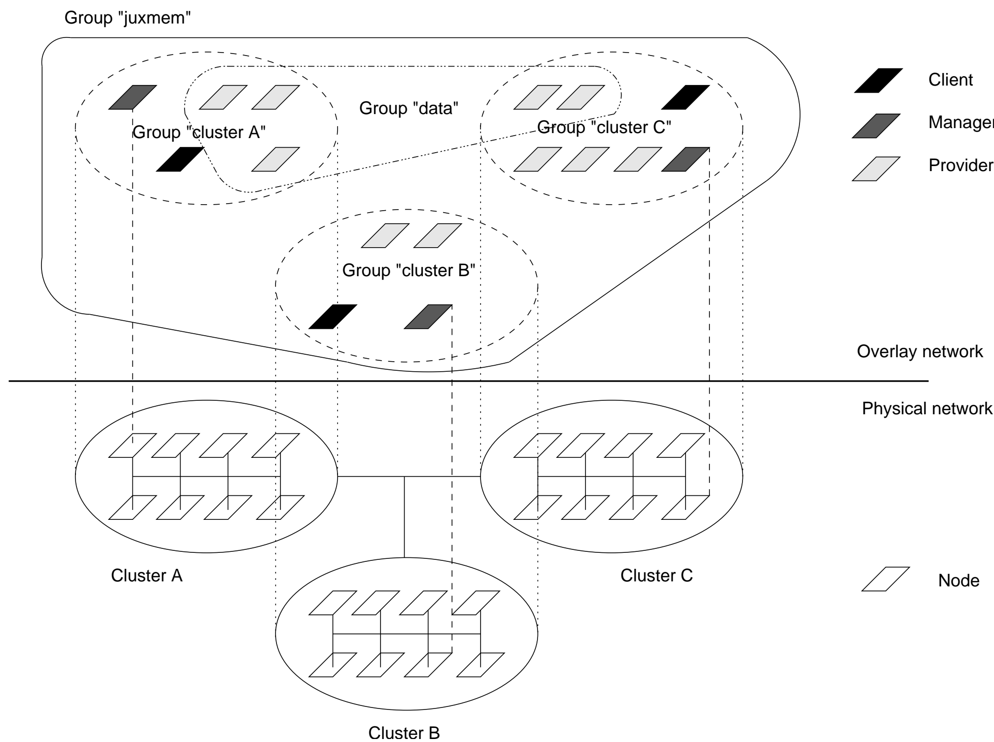


Figure 1: Hierarchy of the entities in the network overlay defined by JUXMEM.

ID can be used by other nodes in order to access existing data through the use of the `juxmem_mmap` function. It is the responsibility of the implementation of the grid data-sharing service to localize the data and perform the necessary data transfers based on this ID. This is how a grid data-sharing service provides a transparent access to data.

To obtain read and/or write access on a data, a process that uses a grid data-sharing service should acquire the lock associated to the data through either `juxmem_acquire` or `juxmem_acquire_read`. This allows the implementation to apply consistency guarantees according to the consistency protocol specified by the user at the allocation time of the data. Note that `juxmem_acquire_read` allows multiple readers to simultaneously access the same data. The `juxmem_attach` function allows locally existing data to be made globally available inside a grid data-sharing service. This function returns a data ID, which is used by other nodes to get access to the data. Finally, `juxmem_unmap` destroys the local copy of a data from a client process, whereas `juxmem_detach` only removes the local copy under the control of a grid data-sharing service.

4 Using JUXMEM for transparent data sharing in the DIET GridRPC middleware

To illustrate how a GridRPC system can benefit from transparent access to data, we have implemented the proposed approach inside the DIET GridRPC middleware, using the JUXMEM data-sharing service. However note that the concept of grid data-sharing service can be used in other GridRPC middleware.

4.1 An overview of a GridRPC middleware framework: DIET

The Distributed Interactive Engineering Toolbox (DIET) platform [10] is a GridRPC middleware, whose architecture is described on Figure 2. It relies on the following entities. *The Client* is an application which

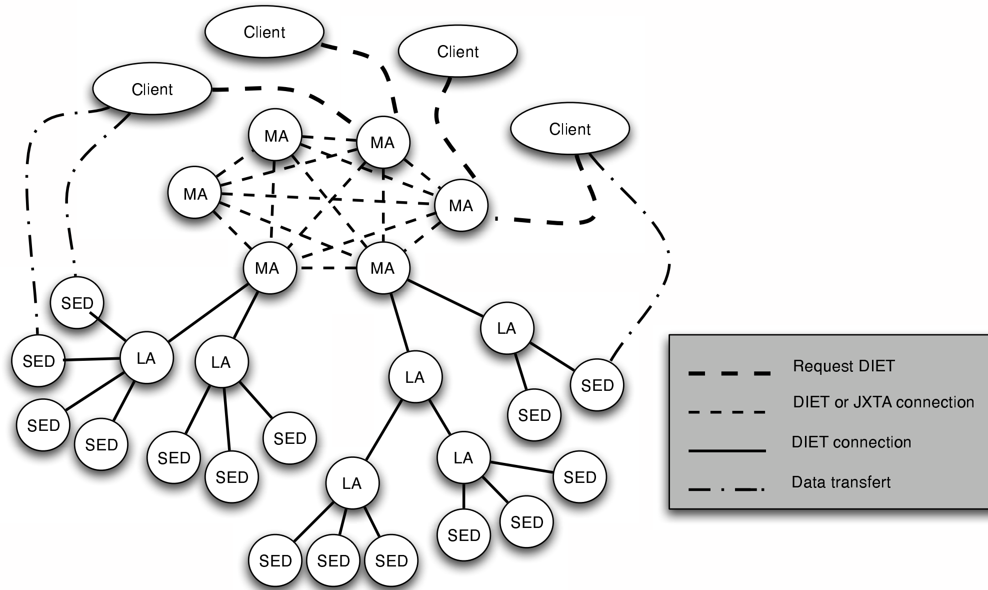


Figure 2: The hierarchical organization of DIET.

uses DIET to solve problems. *Agents (MA or LA)* receive computation requests from clients. A request is a generic description of the problem to be solved with data information (type, size, etc.). Agents collect the computational capabilities of the available servers, and selects the *best* server according to the given request. Eventually, the reference of the selected server is returned to the client, which can then directly submit its request to this server. Note that contrary to other GridRPC middleware and for scalability purpose, agents can be organized in a set of trees forming a forest. *The Server Daemon (SeD)* encapsulates a computational server and makes it available to its parent LA. It also provides the potential clients with an interface for submitting their requests.

Note that, as other GridRPC middleware, DIET specifies three access modes for each data involved in a computation (see section 2.1).

4.2 Our JUXMEM-based data management solution inside DIET

In our work, DIET *internally* uses JUXMEM whenever a data is marked as persistent. However, we distinguish two cases for persistent data, depending on if the DIET client needs to access persistent data at the end of the computation or not. In the former case, the persistent mode is set by the client to `PERSISTENT` and in the later case it is `PERSISTENT_RETURN`.

Listings 1 and 2 show an example of how DIET internally uses JUXMEM to manage data for the multiplication of two matrices A and B. The output of the computation produces the matrix C. Figure 3 presents entities implied in this example: one DIET client D, one DIET SeD $S1$ and two JUXMEM providers $F1$ and $F2$[1]. Let us assume that all matrices are persistent. First, input matrices are attached inside JUXMEM on the client side (step 1 of Figure 3 and lines 4 and 5 of listing 1), and their IDs $ID(A)$ and $ID(B)$ are sent in the computational request to $S1$ (step 2 and line 7). On the server side, these IDs are used to locally map and acquire in read mode matrices (step 3 and lines 2 to 6 of listing 2). Then, the computation is performed and matrix C is produced (line 8). Therefore, the read lock on matrices A and B can be released (lines 10 and 11), and matrix C is attached inside JUXMEM (step 4 and line 12). The ID $ID(C)$ of the matrix C is sent back to the client D (step 5), so that it can be locally mapped and acquired in read mode (steps 6 and 10 and lines 9 to 11 of listing 1).

[1]For the sake of clarity on the figure, however in practice $F2$ can be equal to $F1$.

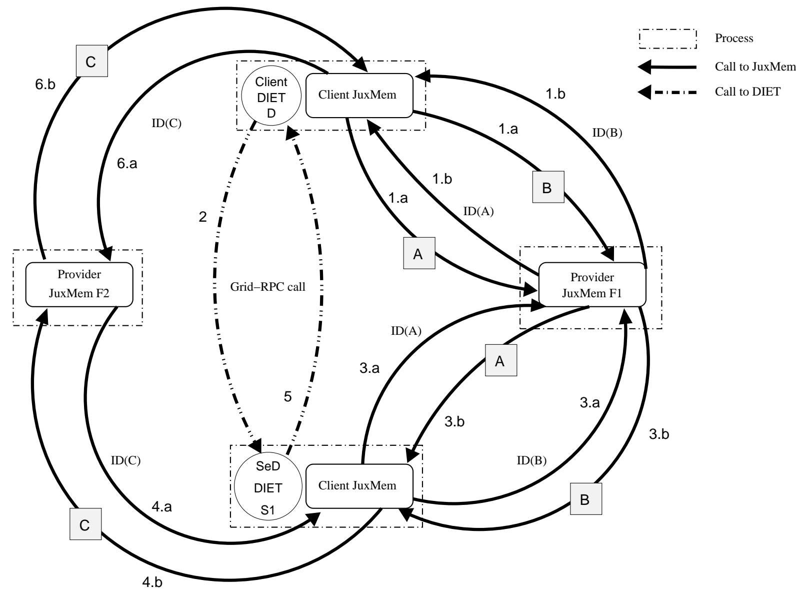


Figure 3: Multiplication of two matrices by a DIET client configured to use JUXMEM for persistent data management.

<table>
<tr><td></td><td colspan="2">Client side</td><td colspan="2">SeD side</td></tr>
<tr><td></td><td>Before computation</td><td>After computation</td><td>Before computation</td><td>After computation</td></tr>
<tr><td>in</td><td>attach;
msync;
detach;</td><td></td><td>mmap;
acquire_read;</td><td rowspan="2">release;
unmap;</td></tr>
<tr><td>inout</td><td>attach;
msync;</td><td>acquire_read;
release;</td><td>mmap;
acquire;</td></tr>
<tr><td>out</td><td></td><td>mmap;
acquire_read;
release;</td><td></td><td>attach;
msync;
unmap;</td></tr>
</table>

Table 1: Use of JUXMEM inside DIET for `in`, `inout` and `out` data on both client/server side, before and after a computation. The `juxmem` prefix has been omitted.

Listing 1: Internal DIET client code related to JUXMEM for the multiplication of two persistent matrices A and B on a SeD.

```
1  grpc_error_t grpc_call (grpc_function_handle_t *handle) {
2    grpc_serveur_t *SeD = request_submission(handle);
3    ...
4    char *idA = juxmem_attach(handle->A, data_sizeof(handle->A));
5    char *idB = juxmem_attach(handle->B, data_sizeof(handle->B));
6    ...
7    char *idC = SeD->remote_solve(multiply, idA, idB);
8    ...
9    juxmem_mmap(handle->C, data_sizeof(handle->C), idC);
10   juxmem_acquire_read(handle->C);
11   juxmem_release(handle->C);
12   ...
13 }
```

Table 1 summarizes the interaction between DIET and JUXMEM in each case, depending on the data access mode (e.g. `in`, `inout`, `out`), before or after the computation was performed and this on both client/server side. In the previous example, matrices A and B are `in` data, and matrix C is an `out` data. Note that in the case of both `inout` and `out` data, after the computation and on the client side, calls to JUXMEM are executed only if the persistent mode is `PERSISTENT_RETURN`.

Listing 2: Internal DIET SeD code related to JuxMem for the multiplication of two persistent matrices A and B on a SeD.

```
1  char* solve (grpc_function_handle_t *handle, char *idA, char *idB) {
2    double *A = juxmem_mmap(NULL, data_sizeof(handle->A), idA);
3    double *B = juxmem_mmap(NULL, data_sizeof(handle->B), idB);
4
5    juxmem_acquire_read(A);
6    juxmem_acquire_read(B);
7
8    double *C = multiply(A, B);
9
10   juxmem_release(A);
11   juxmem_release(B);
12   return idC = juxmem_attach(C, data_sizeof(handle->C));
13 }
```

Modifications performed inside the DIET GridRPC middleware to use JuxMem for the management of persistent data are small. They consist of 200 lines of C++ code, activated whenever DIET is configured to use JuxMem. Consequently, DIET is linked with the C/C++ binding of JuxMem. In our setting, DIET clients or SeDs use the API of JuxMem to store/retrieve data, therefore they are acting as JuxMem clients. Note also that our solution supports GridRPC interoperability, since JuxMem's API is internally used by DIET : no additional functions dealing with data management is added to the client API specification.

5 Experimental evaluations

In this section, we present the experimental evaluation of our JuxMem-based data management solution inside DIET.

5.1 Experimental conditions

We performed tests over 2 clusters (Rennes and Orsay) of the French Grid'5000 testbed [8], using a total number of 53 nodes. Grid'5000 is an experimental grid platform consisting of 9 sites (clusters) geographically distributed in France, whose aim is to gather a total of 5,000 CPUs in the near future. The nodes used for our experiments consist of machines using dual 2.2 GHz AMD Opteron, outfitted with 2 GB of RAM each, and running a 2.6 version Linux kernel; the network layer used is a Giga Ethernet (1 Gb/s) network inside each cluster of Grid'5000. Between clusters, links of 10 Gb/s are used and the latency is 4,5 ms. However, due to default TCP buffers size, the measured bandwidth between Rennes and Orsay is 3.71 MB/s. We kept this small bandwidth in order to emulate a client using a grid to perform its computations.

Tests were executed using JuxMem 0.3 and DIET 2.1. All benchmarks are compiled using `gcc` 4.0 with the `-O2` level of optimization. As regards deployment, we used the ADAGE [15] generic deployment tool for JuxMem and GoDIET [9] for DIET.

5.2 Benefits of persistence: results based on a synthetic service

The goal of this test is to demonstrate and measure the benefits of the management of persistent data by JuxMem, in terms of impact on the overall execution time of a client's series of service invocations. For this test, one client located in one cluster of the grid (here: Rennes) performs a series of 32 asynchronous GridRPC calls to a simple synthetic service. The service is provided by 16 SeDs distributed on 16 nodes located in a second grid cluster (Orsay). Each SeD processes its requests sequentially. The service uses the `sleep` function on the server side. This sleep service requires a single matrix M, in `in` access mode. For each run of the benchmark, we vary the execution time s of the `sleep` function on the server side (s equals to 5 or 60 seconds), as well as the size m of M (from 1 byte to 64 MB). On the JuxMem side, in addition to the 17 JuxMem clients (16 SeDs and 1 DIET client), the JuxMem network simply consists of 1 manager and 1 storage provider. The provider is hosted on the cluster of Orsay.

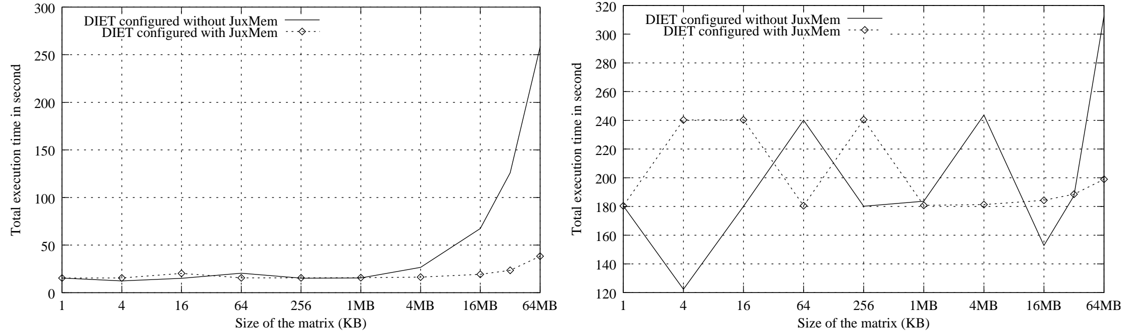


Figure 4: Total execution time (t) of a test with one persitent matrix of variable size when DIET is configured with and without JuxMem. The execution time of the service is s ($s = 5$ seconds on the left and $s = 60$ seconds on the right).

Figure 4 shows the total execution time t of the client code when DIET is configured with and without JuxMem for the management of persistent data ($s = 5$ seconds on the left and $s = 60$ seconds on the right). For $m = 64$ MB, t is equal to 257 seconds when DIET is configured without JuxMem (with $s = 5$ seconds). This time is lowered to 38 seconds when DIET is configured with JuxMem (still with $s = 5$ seconds). This is a speedup of 6.8. It is mainly due to the fact that, at the first call to the sleep service, the matrix M is transfered and stored on the JuxMem provider. Consequently, the client no longer has to send M 32 times. Using JuxMem, this is done only once. Then M is accessed from the provider, located inside the same cluster as the SeDs therefore using high-bandwidth links. The speedup start to became noticeable for values of m higher than 1 MB. Note that very small values of m, the overhead of using JuxMem, instead of directly sending data to SeD, is low (the value of t is essentially the same in both cases). When s is equal to 60 seconds, a smaller speedup can be observed (1.6 for the maximum speedup). This is explained by the fact that execution times of `sleep` functions mainly dominates the total execution time t. The zigzag shape of the curves is due to the SeD's inability to update their availibility information quickly enough, as the 32 asynchronous GridRPC calls are being submitted to the DIET hierarchy. Scheduling decisions are then impacted by this delays.

5.3 Benefits of persistence: results based on a DGEMM service

In our second test, we replace our sleep service by a real computation, using a DGEMM (matrix multiplication) service. The main goal is to validate our initial results of Section 5.2 using a real service, with complex data access patterns. This service requires the BLAS (Basic Linear Algebra Subprograms) library to perform computation over matrices [13]. More precisely, on a SeD this service requires two `in` matrices A and B of `double` type, two `in` parameters (α and β) of `int` type as well as one `inout` matrix C. The service then modifies the matrix C in the following way:

$$C = \alpha \times A \times B + \beta \times C \tag{1}$$

In this test, we vary the size m of A and B (from 1 byte to 32 MB). Note that for this benchmark, values of A and B are unchanged, whereas the value of C is different between each call.

In terms of distributed entities involved, we use 2 settings similar to the one used for the previous test. One client (located in the Rennes cluster) performs 32 asynchronous calls to the DGEMM service (provided by 16 SeDs running in the Orsay cluster). On the JuxMem side, in addition to the 17 JuxMem clients (16 SeDs and and 1 DIET client), the JuxMem storage infrastructure deployed uses two different configurations: 1) 1 manager and 1 provider (noted configuration X) and 2) 1 manager and 34 storage providers (noted configuration Y). Using these two different configurations will allow us to measure the impact of the matrix distribution over a set of providers. Note that in both configurations, storage providers are hosted on the same cluster as the computing server (Orsay).

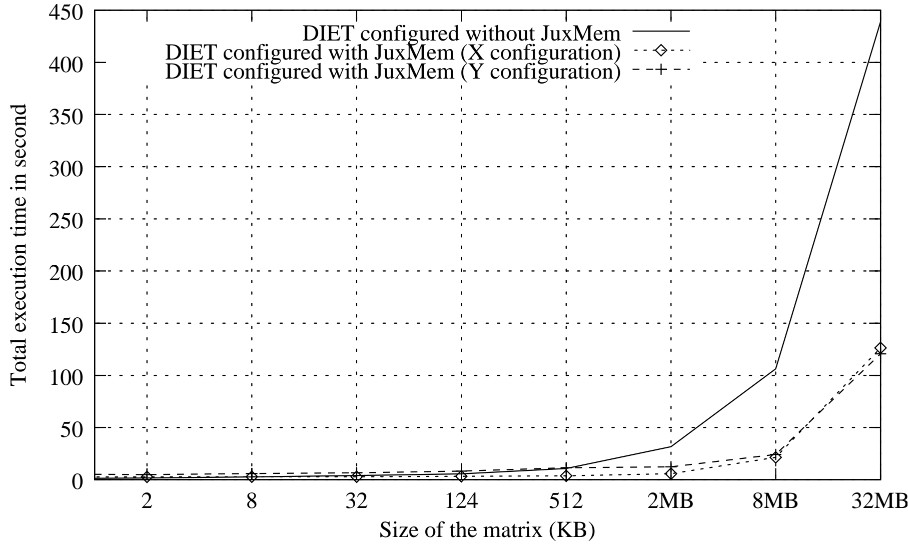


Figure 5: Total execution time (t) of the DGEMM test when DIET is configured with and without JuxMem (using two different configurations of JuxMem X and Y).

The Figure 5 shows the total execution time t of the client code when DIET is configured with and without JuxMem respectively (in both configurations X and Y). For $m = 32$ MB, t is equal to 438 seconds when DIET is configured without JuxMem. This time is lowered to 126 seconds when DIET is configured with JuxMem in the X configuration. When JuxMem is deployed in the Y configuration, t equals 120 seconds. This is a speedup of 3.4 for the X configuration of JuxMem and 3.6 for the Y configuration. As in the previous benchmark (see section 5.2), it is mainly due to the fact that at the first GridRPC call, matrices A and B are transfered and stored on the JuxMem provider(s), hosted on the same cluster as the available services are located. Consequently, the client no longer has to send these matrices A and B 32 times. Using JuxMem, this is done only once. Then matrices A and B are accessed from inside the same cluster therefore using high-bandwidth links. The speedup start to became noticeable for values of m higher than 2 MB. Note that for higher value of m, the Y configuration of JuxMem slightly improves t, compared to the X configuration. This is explained by the load balancing enabled by the Y configuration, as matrices are distributed over the set of providers. However for small matrices, it is the opposite: t is higher when JuxMem is deployed in the Y configuration and in the X one. This is explained by the higher latency of performing a GridRPC call, as matrices have to be retrieved from different providers.

6 Conclusion

Programming grid infrastructures remains a significant challenge. The GridRPC model is the grid form of the classical RPC approach. It offers the ability to perform asynchronous coarse-grained parallel tasking, and hides the complexity of server-level parallelism to clients. In its current state, the GridRPC model has not specified adequate mechanisms for efficient data management. One important issue regards data persistence, as multiple GridRPC calls with data dependencies are executed. In this paper, we propose to couple the GridRPC model with a *transparent data access model*. Such a model is provided by the concept of *grid data-sharing service*. Data management (persistent storage, transfer, consistent replication) is totally delegated to the service, whereas the applications simply access data via global identifiers. The service automatically localizes and transfers or replicates the data as needed.

We have illustrated our approach by showing how the DIET GridRPC middleware can benefit from the above properties by using the JuxMem grid data-sharing service. Experimental measurements on the Grid'5000 testbed show that introducing persistent storage has a clear impact on the execution time. Over

a scenario performing matrix computation, speedups of over 3 can be observed when JuxMem is used to provide data persistence.

To extend our contributions to data management in NES through the features offered by JuxMem, further research can be performed. First, current work is in progress with the goal to benchmark the benefits of the use of JuxMem on the execution time of the Grid-TLSE application, a complex sparse matrix manipulation portal. Second, the implementation of a classical file-system API over JuxMem would allow applications based on this API to transparently leverage JuxMem's functionalities. We have already started such a work, called JuxMemFS, by relying on the FUSE library [23] available on Linux systems. We also plan to provide data placement information to the request scheduling algorithms. This would make it possible to balance more precisely the load among available servers. Finally, it would be interesting to evaluate the impact of using the fault tolerance mechanisms provided by JuxMem (not discussed in this paper), in presence of data storage failures.

References

[1] Patrick R. Amestoy, Iain S. Duff, Jean-Yves L'Excellent, and Jacko Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[2] Gabriel Antoniu, Marin Bertier, Eddy Caron, Frédéric Desprez, Luc Bougé, Mathieu Jan, Sébastien Monnet, and Pierre Sens. GDS: An Architecture Proposal for a grid Data-Sharing Service. In V. Getov, D. Laforenza, and A. Reinefeld, editors, *Future Generation Grids*, CoreGRID series, pages 133–152. Springer, November 2005.

[3] Gabriel Antoniu, Luc Bougé, and Mathieu Jan. JuxMem: An Adaptive Supportive Platform for Data Sharing on the Grid. *Scalable Computing: Practice and Experience*, 6(3):45–55, November 2005.

[4] Gabriel Antoniu, Jean-François Deverge, and Sébastien Monnet. How to bring together fault tolerance and data consistency to enable grid data sharing. *Concurrency and Computation: Practice and Experience*, 18(13):1705–1723, November 2006.

[5] Dorian Arnold, Sudesh Agrawal, Susan Blackford, Jack Dongarra, Micelle Miller, Kiran Sagi, Zhiao Shi, and Sthish Vadhiyar. Users' Guide to NetSolve V1.4. Computer Science Dept. Technical Report CS-01-467, University of Tennessee, Knoxville, TN, July 2001.

[6] Dorian C. Arnold, Dieter Bachmann, and Jack Dongarra. Request Sequencing: Optimizing Communication for the Grid. In *Proc. of the 6th Intl. Euro-Par Conference (Euro-Par 2000)*, volume 1900 of *Lecture Notes in Computer Science*, pages 1213–1222, Munich, Germany, August 2000. Springer.

[7] Alessandro Bassi, Micah Beck, Graham Fagg, Terry Moore, James Plank, Martin Swany, and Rich Wolski. The Internet Backplane Protocol: A Study in Resource Sharing. *Future Generation Computer Systems*, 19(4):551–562, 2003.

[8] Franck Cappello, Eddy Caron, Michel Dayde, Frederic Desprez, Emmanuel Jeannot, Yvon Jegou, Stephane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, and Olivier Richard. Grid'5000: A Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform. In *Proc. of the 6th IEEE/ACM Intl. Workshop on Grid Computing (Grid '05)*, pages 99–106, Seattle, Washington, USA, November 2005.

[9] Eddy Caron, Pushpinder Kaur Chouhan, and Holly Dail. GoDIET: A Deployment Tool for Distributed Middleware on Grid'5000. In *Workshop on Experimental Grid Testbeds for the Assessment of Large-Scale Distributed Applications and Tools (EXPGRID)*, pages 1–8, Paris, France, June 2006. In conjunction with 15th IEEE Intl. Symposium on High Performance Distributed Computing (HPDC 15), IEEE Compuer Society.

[10] Eddy Caron and Frédéric Desprez. DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid. *Intl. Journal of High Performance Computing Applications*, 20(3):335–352, 2006.

[11] Michel Daydé, Luc Giraud, Montse Hernandez, Jean-Yves L'Excellent, Chiara Puglisi, and Marc Pantel. An Overview of the GRID-TLSE Project. In *Poster Session of 6th Intl. Meeting (VECPAR '04)*, pages 851–856, Valencia, Espagne, June 2004.

[12] Frédéric Desprez and Emmanuel Jeannot. Improving the GridRPC Model with Data Persistence and Redistribution. In *Proc. of the 3rd Intl. Symposium on Parallel and Distributed Computing (ISPDC '2004)*, pages 193–200, Cork, Ireland, July 2004. IEEE Computer Society.

[13] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, 1990.

[14] Bruno Del Fabbro, David Laiymani, Jean-Marc Nicod, and Laurent Philippe. Data management in grid applications providers. In *Proc. of the 1st Intl. Conference on Distributed Frameworks for Multimedia Applications (DFMA '05)*, pages 315–322, Besançon, France, February 2005.

[15] Sébastien Lacour, Christian Pérez, and Thierry Priol. Generic application description model: Toward automatic deployment of applications on computational grids. In *Proc. of the 6th IEEE/ACM Intl. Workshop on Grid Computing (Grid 2005)*, pages 4–8, Seattle, WA, USA, November 2005. Springer.

[16] Hidemoto Nakada, Yoshio Tanaka, Keith Seymour, Frédéric Desprez, and Craig Lee. The End-User and Middleware APIs for GridRPC. In *Proc. of the Workshop on Grid Application Programming Interfaces (GAPI '04)*, Brussels, Belgium, September 2004. In conjunction with Global Grid Forum 12 (GGF).

[17] Yoshihiro Nakajima, Mitsuhisa Sato, Taisuke Boku, Daisuke Takahasi, and Hitoshi Gotoh. Performance Evaluation of OmniRPC in a Grid Environment. In *Proc. of the 2004 Intl. Symposium on Application and the Internet Workshops (SAINTW '04)*, pages 658–665, Tokyo, Japan, January 2004. IEEE Computer Society.

[18] Mitsuhisa Sato, Taisuke Boku, and Daisuke Takahasi. OmniRPC: a Grid RPC System for Parallel Programming in Cluster and Grid Environment. In *Proc. of the 3rd IEEE/ACM Intl. Symposium on Cluster Computing and the Grid (CCGrid '03)*, pages 206–213, Tokyo, Japan, May 2003. IEEE Computer Society.

[19] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee, and Henri Casanova. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In Manish Parashar, editor, *Proc. of the 3rd Intl. Workshop on Grid Computing (GRID '02)*, volume 2536 of *Lecture Notes in Computer Science*, pages 274–278, Baltimore, MD, USA, November 2002. Springer.

[20] Yoshio Tanaka, Hiroshi Takemiya, Hidemoto Nakada, and Satoshi Sekiguchi. Design, implementation and performance evaluation of gridRPC programming middleware for a large-scale computational grid. In *Proc. of the 5th Intl. Workshop on Grid Computing (GRID 2004)*, pages 298–305, Pittsburgh, PA, USA, 2004. IEEE Computer Society.

[21] Bernard Traversat, Ahkil Arora, Mohamed Abdelaziz, Mike Duigou, Carl Haywood, Jean-Christophe Hugly, Eric Pouyoul, and Bill Yeager. Project JXTA 2.0 Super-Peer Virtual Network. http://www.jxta.org/project/www/docs/JXTA2.0protocols1.pdf, May 2003.

[22] Asim YarKhan, Keith Seymour, Kiran Sagi, Zhiao Shi, and Jack Dongarra. Recent Developments in GridSolve. *Intl. Journal of High Performance Computing Applications*, 20(1):131–141, 2006.

[23] Filesystem in Userspace (FUSE). http://fuse.sourceforge.net/.

[24] Grid Remote Procedure Call WG (GRIDRPC-WG). https://forge.gridforum.org/projects/gridrpc-wg/.