Towards a Transparent Data Access Model for the GridRPC Paradigm

Gabriel Antoniu, Eddy Caron, Frédéric Desprez, Aurélia Fèvre, Mathieu Jan

Towards a Transparent Data Access Model for the GridRPC Paradigm

Gabriel Antoniu*, Eddy Caron†, Frédéric Desprez†, Aurélia Fèvre†, Mathieu Jan‡

*INRIA/IRISA, Campus de Beaulieu, 35042 Rennes, France
Gabriel.Antoniu@irisa.fr
†LIP/ENS-Lyon/INRIA, 46 Allée d'Italie, 69364 Lyon, France
{Eddy.Caron,Frederic.Desprez,Aurelia.Fevre}@ens-lyon.fr
‡LRI/INRIA, University of Paris South, 91405 Orsay, France
Mathieu.Jan@lri.fr

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. 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 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. Notably, our experiments performed on the Grid'5000 using a real-life application show the efficiency of using JUXMEM for managing persitent data in the GridRPC model: application execution times in a grid environment are of the same order as in a cluster environment.

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. Multi-parametric applications, for instance, which consist in applying the same algorithm to different input data, can benefit from an efficient use of grid computing infrastructures.

Running such applications on large-scale grid infrastructures requires the use of adequate programming paradigms. The *Grid Remote Procedure Call* (GridRPC) [1] approach provides such a paradigm, which extends the classical RPC model by enabling asynchronous, coarse-grained parallel tasking. GridRPC seems to be a good approach

‡ This author's work has mainly been done at INRIA/IRISA.

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 to use among a large set of candidates.

A team of researchers of the Global Grid Forum (GGF [1]) has defined a standard API for the GridRPC paradigm [2]. However, in this specification, data management has been left as an open (although fundamental) issue. For instance, data transfer in the distributed environment is left to the user, who must explicitly move them back and forth between clients and servers. This clearly increases the program complexity, 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 [3] GridRPC middleware and the JUXMEM [4] grid data-sharing service. We evaluate our approach through experiments realized on the Grid'5000 [5] testbed.

The remainder of the paper is organized as follows. Section 2 introduces the GridRPC model, presents the requirements of a sample application with respect to data management, then 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 using a real-life application. 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 Open Grid Forum (OGF), 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 [6] working group of the GGF. The goal of this group is to specify the syntax and the programming interface at the client level [1]. 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 multiple applications to concurrently run on a shared set of grid resources. Examples of middleware that implement the GridRPC specification are Ninf-G [7], NetSolve [8], GridSolve [9], DIET [3], and OmniRPC [10].

[1] GGF, recently merged with EGA (Enterprise Grid Alliance) to create the OGF (Open Grid Forum)

Note that the GridRPC model knows the target server. Nevertheless some GridRPC middleware proposes to discover the best server automatically. In this case, *servers* register *services* to a *directory*. To invoke a service, instead of using the server given by GridRPC call function, *clients* bypass this parameter and look for a suitable, possibly "the best" server according to some performance metric. This selection is made out of a set of candidates proposed by the directory. GridRPC does not define any standard for the underlying resource discovery mechanism. The server selection is performed by one or several *agents* pr *schedulers*. The decision is usually made 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: available services, server load, input data location, etc. Based on this information, the agents make their decisions so as 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 statistics from runs of various sophisticated sparse matrix algorithms on specific data. The input data are either submitted by the user, or picked up from a 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 [12], over a matrix, with fixed parameters. We focus on the MUMPS solver for our experiments (see Section 5.2).

When using Grid-TLSE, 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 evaluation of this 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 transfer 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 application 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.

Passing arguments 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. Consequently, data consistency must be guaranteed in case of concurrent accesses.

Transparent data localization and transfer. Such a transparency would simplify the use of the GridRPC paradigm at a large scale, as developpers would no longer need to explicitly move data.

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

GridRPC interoperability. Any solution addressing the previous 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 OGF, data persistence is not yet 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* [13]. 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 [14] 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 [15]). 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 [16]. The user has to manually define a initialization procedure to indicate which input data should be sent and stored prior to computations. 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 [17] on the computing servers. However, as in both cases ad-hoc solutions are used to handle data persistence, GridRPC interoperability cannot be guaranteed, 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 within the framework of the GridRPC working group of the OGF [2]. 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, transparent access to data is also envisioned. However, replication, consistency guarantees and fault tolerance issues have not been addressed yet.

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 replication and/or transfer among clients, computing servers and storage servers as needed. It should also transparently provide fault tolerance and data consistency guarantees in such dynamic, large-scale, distributed environments.

In order to achieve a real virtualization of the management of large-scale distributed data, a step forward has been made by the proposal of a *transparent data access model*, as a part of the concept of *grid data-sharing service* [18]. 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). An illustration of this concept has been realized through the JUXMEM software experimental platform [4]. The service specification includes three main properties.

Persistence. The data sharing service provides persistent data storage and allows 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, based on previously defined *consistency models and protocols*.

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

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 show how *persistence* can be provided in a transparent way. *Data consistency* is ensured by providing a multi-protocol framework allowing various consistency models and protocols to be implemented. JUXMEM currently supports the *entry consistency* model through a hierarchical, *fault-tolerant* protocol. A description of the concepts and technical details related to data consistency and fault tolerance is beyond the focus of this paper. The corresponding mechanisms have been detailed in [19].

3.2 Overview of the JUXMEM data-sharing service

The JUXMEM [4] software experimental platform illustrates the concept of data-sharing service. 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 it is therefore expressed in terms of *hierarchical* groups. The goal is to accurately take into account the latency hierarchy of the physical network topology, to take advantage of the low-latency links within the clusters and reduce higher-latency, inter-cluster communications. All nodes participating to the data-sharing service network overlay are members of the JUXMEM group. All members of the JUXMEM group that belong to the same physical cluster form a *cluster group*.

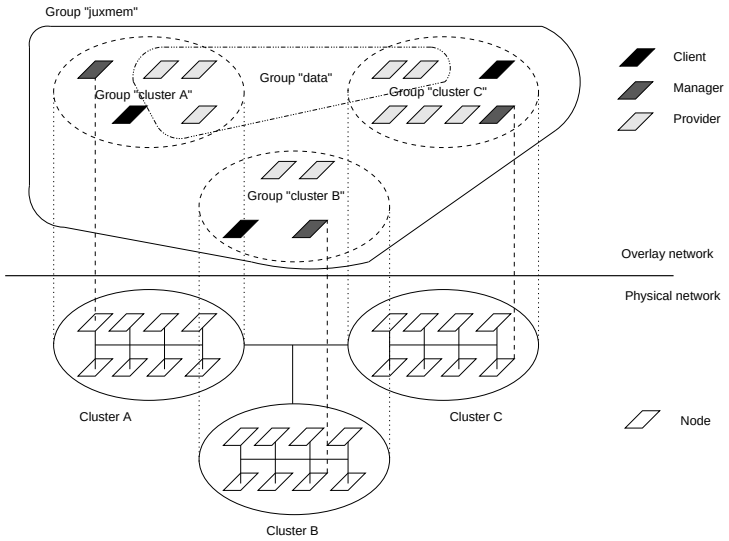

Fig. 1. Hierarchy of the entities in the network overlay defined by JUXMEM.

Any cluster group consists of *provider* nodes which supply memory for data storage. Each cluster group is managed by a special peer, called a *manager*. Managers make up the backbone of a given JUXMEM overlay 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 [19]. This software architecture has been implemented using the JXTA [20] generic P2P platform.

3.3 JUXMEM from the user's perspective

The programming interface proposed by 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 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.

According to the entry consistency model implemented by JUXMEM, processes that need to access data need to properly synchronize by acquiring a lock associated to that data. This is done by calling `juxmem_acquire_read` (prior to a read access) or `juxmem_acquire` (prior to a write access). Note that `juxmem_acquire_read` allows multiple readers to simultaneously access the same data. The `juxmem_release` primitive must be called after the access, to release the lock. These synchronization primitives allow the implementation to provide consistency guarantees according to the consistency protocol specified by the user at the allocation time of the data.

To make local data globally available in JUXMEM, the `juxmem_attach` function can be used. This function creates the corresponding data replicas (similarly to the `juxmem_malloc` primitive) and returns a data ID which is used by other nodes to get access to the data. When they no longer need to access the shared data, clients can remove their local data copies using the `juxmem_unmap` primitive. Finally, to keep the local data copy while removing it from the control of the grid data-sharing service, clients must use the `juxmem_detach` primitive.

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. Note however that the concept of grid data-sharing service can also be used in connection with other GridRPC middleware.

4.1 An overview of a GridRPC middleware framework: DIET

The Distributed Interactive Engineering Toolbox (DIET) platform [3] is a GridRPC middleware, whose architecture is described on Figure 2. It relies on the following en-

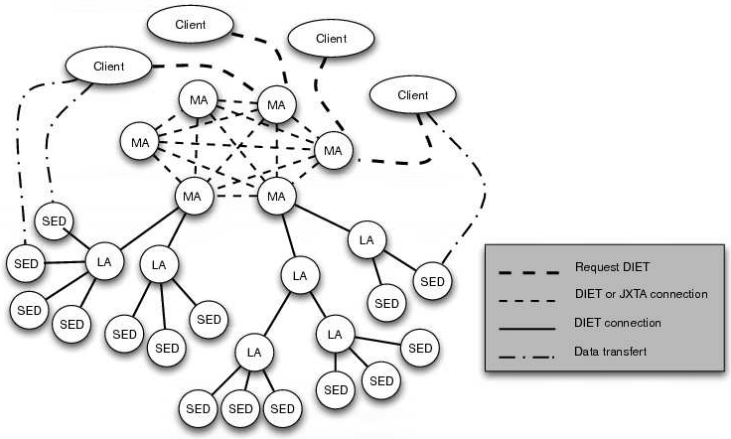

Fig. 2. The hierarchical organization of DIET.

tities. A *Client* is an application which uses DIET to solve problems. *Agents* 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. As opposed to other GridRPC middleware, for scalability purpose, agents can be organized in a set of trees forming a forest of local agents (LA) rooted at a master agent (MA). *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.

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

4.2 How DIET uses JUXMEM to manage data

In our work, DIET *internally* uses JUXMEM whenever a data is marked as persistent. However, we distinguish two cases for persistent data. If the DIET client needs to access persistent data at the end of the computation, the persistence mode is set to `PERSISTENT_RETURN`. Otherwise, it is set to `PERSISTENT`.

Listings 1.1 and 1.2 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 the entities involved: one DIET client D, one DIET SeD $S1$ and two JUXMEM providers $F1$ and $F2$[2]. Let us assume that all matrices are persistent. First, input matrices are stored into JUXMEM by the client (step 1 of Figure 3, lines 5 and 6 of Listing 1.1), and their IDs $ID(A)$ and $ID(B)$ are sent in the computational request to $S1$ (step 2, line 8). On the server side, these IDs are used to locally map and acquire the input matrices in read mode (step 3, lines 5 to 8 of Listing 1.2). Then, the computation produces matrix C (line 10). Therefore, the read lock on matrices A and B is released (lines 12 and 13), and matrix C is attached inside JUXMEM (step 4 and line 14). Its identifier ($ID(C)$) is sent back to the client D (step 5),

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

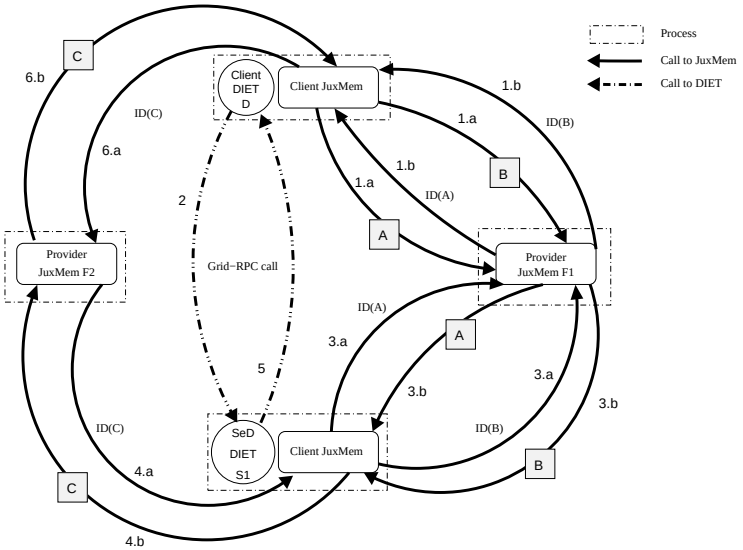

Fig. 3. Multiplication of two matrices by a DIET client configured to use JUXMEM for persistent data management.

so that it can be locally mapped and acquired in read mode by the client (steps 6, lines 10 to 12 of Listing 1.1).

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

Listing 1.1. Internal DIET client code related to JUXMEM for the multiplication of two persistent matrices A and B on a SeD.

```
1  char*
2  solve (grpc_function_handle_t *handle,
3         char *idA, char *idB) {
4    ...
5    double *A = juxmem_mmap(NULL, data_sizeof(handle->A), idA);
6    double *B = juxmem_mmap(NULL, data_sizeof(handle->B), idB);
7    juxmem_acquire_read(A);
8    juxmem_acquire_read(B);
9    ...
10   double *C = multiply(A, B);
11   ...
12   juxmem_release(A);
13   juxmem_release(B);
14   return idC = juxmem_attach(C, data_sizeof(handle->C));
15 }
```

Listing 1.2. Internal DIET SeD code related to JUXMEM for the multiplication of two persistent matrices A and B on a SeD.

Table 4.2 summarizes the interaction between DIET and JUXMEM in each case, depending on the data access mode (e.g. `in`, `inout`, `out`) 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 for `inout` and `out` data, calls to JUXMEM are executed after the computation on the client side only if the persistent mode is `PERSISTENT_RETURN`.

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

	Client side		SeD side	
Computation	Before	After	Before	After
in	`attach; msync; detach;`		`mmap; acquire_read;`	`release; unmap;`
inout	`attach; msync;`	`acquire_read; release;`	`mmap; acquire;`	
out		`mmap; acquire_read; release;`		`attach; msync; unmap;`

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 JUXMEM's API to store/retrieve data, thereby acting as JUXMEM clients. Note also that our solution supports GridRPC interoperability, DIET simply uses JUXMEM's API, with no extra code for data management.

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 using 4 clusters (Rennes, Orsay, Toulouse and Lyon) of the French Grid'5000 testbed [5], using a total number of 3 sites simultaneously for a total number of 129 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, 2.4, 2.6 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 ranges from 4,5 ms to 10 ms.

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 [21] generic deployment tool for JUXMEM and GoDIET [22] for DIET.

5.2 Experiments using MUMPS: a sparse parallel solver

Our goal 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 real-life application. We focus on `MUMPS` ("MUltifrontal Massively Parallel Solver"), a package for solving systems of linear equations of the form $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a square sparse matrix that can be either asymmetric, symmetric positive definite, or general symmetric. `MUMPS` uses a multifrontal technique which is a direct method based on either the LU or the LDL^T factorization of the matrix. We refer the reader to the paper [23] for full details of these techniques.

We performed 3 sets of experiments using MUMPS (noted $E1$, $E2$ and $E3$). $E1$ has been performed in a one-cluster environment, whereas $E2$ and $E3$ are performed in a multi-cluster environment, using the Grid'5000 testbed, with $E3$ at a larger scale compared to $E2$. In all our experiments, a client loops 32 times on a synchronous GridRPC call to MUMPS, with the aforementioned linear equation to solve. A and b are input data (data access mode set to `in`) whereas x is an output data. Between each GridRPC call, b is changed according to the resolution needs while A is unchanged and its persistence mode is set to `PERSISTENT`. Therefore note that when JUXMEM is used by DIET for the management of matrices A, DIET calls JUXMEM primitives according to the first row of Table 4.2. For all experiments we used 2 different sizes for the $\mathbf{A}$ matrix: a medium size of 22 MB ($A1$) and a larger size of 52 MB ($A2$). Let us stress the difficulty of setting up such kind of experiments for a real-life, complex application, using an environment which relies on 2 different runtime software (JUXMEM and DIET), based on different technologies (JXTA and CORBA respectively) and using different deployment tools that need to interact with each other.

As a first experiment performed inside a single Grid'5000 cluster, we simply deployed a DIET hierarchy made of 1 MA, 1 LA and 1 SeD, as well as a JUXMEM network made of 1 provider and 1 manager. The goal of this experiment $E1$ is to measure the overhead of using JUXMEM for data management. Results show that if DIET is configured to use JUXMEM to store persistent data, the total execution time of all calls slightly increases, compared to DIET configured without JUXMEM: from 36.6 to 41.3 seconds with $A1$ and from 957 to 961 seconds with $A2$. We can argue that the overhead of using JUXMEM for data management of large matrices inside DIET is therefore low: it is less than 1 % with $A2$. Note however that this overhead increases when using smaller matrices, e.g. it reaches 13 % for matrix $A1$.

In a second experiment, our goal is to measure the (expected!) benefits of using JUXMEM for transparently managing persistent data in grid environment (however at a small scale). To do this, we deployed a 3-cluster configuration. In each cluster, we deploy a DIET hierarchy made of 1 LA and 1 SeD and a JUXMEM network made of 1 provider and 1 manager. We use 3 clusters of the Grid'5000 testbed, namely Lyon, Toulouse and Rennes. Results show a clear advantage to use JUXMEM, as the total execution time of the application is reduced by 42 % with $A1$ and by 38 % with $A2$, compared to results obtained for DIET configured without JUXMEM (see Table 5.2). Compared to the $E1$ experiment, the smaller execution time with $A2$, when JUXMEM is used, is explained by the difference of processor performance on the three sites that are used for the computations.

Finally, we performed a third experiment ($E3$) similar to $E2$, where we increased the configuration sizes, by using 32 SeDs and 8 JUXMEM providers in each cluster. The goal of this experiment is to measure the impact of an increasing number of SeDs on the performance of JUXMEM (as this leads to an increasing number of JUXMEM clients

accessing the data). With $A1$, the total execution time is the same for both configurations (DIET configured with or without JUXMEM): 103 seconds. With the larger $A2$ matrix, this time is reduced by 38 % when JUXMEM is used (843 seconds), compared to DIET configured without JUXMEM (1358 seconds). Note that these results are averaged based on 4 runs, since the DIET agent may take different scheduling decisions by choosing different nodes (and clusters) from one computation to another. This also explains the difference between results obtained for $A2$ in this experiment and experiment $E2$: the number of GridRPC calls performed on 1 site may change between runs.

Table 2. Total execution time in seconds of a MUMPS application when DIET is configured to use JUXMEM or not, for 2 different matrices $A1$ and $A2$.

Matrix	$A1$		$A2$	
DIET configured	Without JUXMEM	With JUXMEM	Without JUXMEM	With JUXMEM
Experiment $E1$	36.6	41.3	957	961
Experiment $E2$	92.6	53.7	1420	880
Experiment $E3$	103	103	1358	843

Finally, Table 5.2 summarizes obtained results for experiments $E1$, $E2$ and $E3$ based on MUMPS. Notably, these results demonstrate the advantage to use JUXMEM for managing persistent data in the GridRPC model in a grid environment: the execution time of a MUMPS application is kept to its value as in a cluster execution, despite the high-latency WAN connections. As explained previously, the reduced times (rows 2 and 3 of last columns of Table 5.2) come from the difference in processor speeds of nodes used for the various computations across the sites.

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. Using a real-life application based on a sparse matrix parallel solver, experiments performed on the Grid'5000 testbed show that the use of JUXMEM allows to keep an execution time as if the application was executed in a cluster, despite the high-latency WAN connections.

The main contribution of our approach compared to related work having dealt with data persistence in GridRPC environments consists in showing that efficiency

can be obtained through the use of a *generic* data-sharing service, providing *location-transparent* data access. Moreover, our approach also allows to transparently benefit from replica consistency and fault-tolerance mechanisms. We did not develop these aspects in this paper (they have been illustrated in [19] in a more general way). A GridRPC-specific study of these aspects could be addressed in future work. For instance, it would be interesting to evaluate the impact of using the fault tolerance mechanisms provided by JUXMEM on application execution time, in presence of data storage failures.

To extend our contributions to data management in NES through the features offered by JUXMEM, several directions can be pursued. First, we 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. Then, we would like to implement a cache mechanism inside JUXMEM clients to avoid fetching data from providers at each GridRPC call. In addition, JUXMEM consistency and fault-tolerance mechanisms have been tested using synthetic benchmarks [19], outside the GridRPC model. We would like to further evaluate them by using other real-life DIET applications which exhibit such requirements, such as climate modeling and cosmology simulations. Besides, we also plan to compare JUXMEM with non location-transparent data access solutions, such DIET DTM for instance. Finally, 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 [24] available on Linux systems.

Acknowledgements

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see `https://www.grid5000.fr/`. The work has been partially supported by the GDS (ACI MD) grant and by the LEGO (ANR-CICG05-11) grant.

References

1. Seymour, K., Nakada, H., Matsuoka, S., Dongarra, J., Lee, C., Casanova, H.: Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In Parashar, M., ed.: Proc. of the 3rd Intl. Workshop on Grid Computing (GRID '02). Volume 2536 of Lecture Notes in Computer Science., Baltimore, MD, USA, Springer (2002) 274–278
2. Nakada, H., Tanaka, Y., Seymour, K., Desprez, F., Lee, C.: The End-User and Middleware APIs for GridRPC. In: Proc. of the Workshop on Grid Application Programming Interfaces (GAPI '04), Brussels, Belgium (2004) In conjunction with Global Grid Forum 12 (GGF).
3. Caron, E., Desprez, F.: DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid. Intl. Journal of High Performance Computing Applications **20**(3) (2006) 335–352
4. Antoniu, G., Bougé, L., Jan, M.: JuxMem: An Adaptive Supportive Platform for Data Sharing on the Grid. Scalable Computing: Practice and Experience **6**(3) (2005) 45–55
5. Bolze, R., Cappello, F., Caron, E., Daydé , M., Desprez, F., Jeannot, E., Jégou, Y., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Quetier, B., Richard, O., Talbi, E.G., Irena, T.: Grid'5000: a large scale and highly reconfigurable experimental grid testbed. International Journal of High Performance Computing Applications **20**(4) (2006) 481–494
6. Lee, C., Nakada, H., Tanimura, Y.: Grid Remote Procedure Call WG (GRIDRPC-WG). `https://forge.gridforum.org/projects/gridrpc-wg/` (2003)

7. Tanaka, Y., Takemiya, H., Nakada, H., Sekiguchi, S.: 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), Pittsburgh, PA, USA, IEEE Computer Society (2004) 298–305
8. Arnold, D., Agrawal, S., Blackford, S., Dongarra, J., Miller, M., Sagi, K., Shi, Z., Vadhiyar, S.: Users' Guide to NetSolve V1.4. Computer Science Dept. Technical Report CS-01-467, University of Tennessee, Knoxville, TN (2001)
9. YarKhan, A., Seymour, K., Sagi, K., Shi, Z., Dongarra, J.: Recent Developments in GridSolve. Intl. Journal of High Performance Computing Applications **20**(1) (2006) 131–141
10. Sato, M., Boku, T., Takahasi, D.: OmniRPC: a Grid RPC System for Parallel Programming in Cluster and Grid Environment. In: Proc. of the 3rd IEEE/ACM Intl. Symp. on Cluster Computing and the Grid (CCGrid '03), Tokyo, Japan, IEEE Computer Society (2003) 206–213
11. Daydé, M., Giraud, L., Hernandez, M., L'Excellent, J.Y., Puglisi, C., Pantel, M.: An Overview of the GRID-TLSE Project. In: Poster Session of 6th Intl. Meeting on VECPAR '04, Valencia, Espagne (2004) 851–856
12. Amestoy, P.R., Duff, I.S., L'Excellent, J.Y., Koster, J.: A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM Journal on Matrix Analysis and Applications **23**(1) (2001) 15–41
13. Arnold, D.C., Bachmann, D., Dongarra, J.: 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., Munich, Germany, Springer (2000) 1213–1222
14. Desprez, F., Jeannot, E.: Improving the GridRPC Model with Data Persistence and Redistribution. In: Proc. of the 3rd Intl. Symp. on Parallel and Distributed Computing (ISPDC '2004), Cork, Ireland, IEEE Computer Society (2004) 193–200
15. Bassi, A., Beck, M., Fagg, G., Moore, T., Plank, J., Swany, M., Wolski, R.: The Internet Backplane Protocol: A Study in Resource Sharing. Future Generation Computer Systems **19**(4) (2003) 551–562
16. Nakajima, Y., Sato, M., Boku, T., Takahasi, D., Gotoh, H.: Performance Evaluation of OmniRPC in a Grid Environment. In: Proc. of the 2004 Intl. Symp. on Application and the Internet Workshops (SAINTW '04), Tokyo, Japan, IEEE Computer Society (2004) 658–665
17. Fabbro, B.D., Laiymani, D., Nicod, J.M., Philippe, L.: Data management in grid applications providers. In: Proc. of the 1st Intl. Conference on Distributed Frameworks for Multimedia Applications (DFMA '05), Besançon, France (2005) 315–322
18. Antoniu, G., Bertier, M., Caron, E., Desprez, F., Bougé, L., Jan, M., Monnet, S., Sens, P.: GDS: An Architecture Proposal for a grid Data-Sharing Service. In Getov, V., Laforenza, D., Reinefeld, A., eds.: Future Generation Grids. CoreGRID series. Springer (2005) 133–152
19. Antoniu, G., Deverge, J.F., Monnet, S.: How to bring together fault tolerance and data consistency to enable grid data sharing. Concurrency and Computation: Practice and Experience **18**(13) (2006) 1705–1723
20. Traversat, B., Arora, A., Abdelaziz, M., Duigou, M., Haywood, C., Hugly, J.C., Pouyoul, E., Yeager, B.: Project JXTA 2.0 Super-Peer Virtual Network. `http://www.jxta.org/project/www/docs/JXTA2.0protocols1.pdf` (2003)
21. Lacour, S., Pérez, C., Priol, T.: 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), Seattle, WA, USA, Springer (2005) 4–8
22. Caron, E., Chouhan, P.K., Dail, H.: 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), Paris, France, In conjunction with 15th IEEE International Symposium on High Performance Distributed Computing (HPDC 15), IEEE Compuer Society (2006) 1–8
23. Amestoy, P.R., Guermouche, A., L'Excellent, J.Y., Pralet, S.: Hybrid scheduling for the parallel solution of linear systems. Parallel Computing **32**(2) (2006) 136–156
24. Szeredi, M.: Filesystem in Userspace (FUSE). `http://fuse.sourceforge.net/` (2004)