Improving the GridRPC Model with Data Persistence and Redistribution

Frédéric Desprez
LIP, INRIA-ENS-Lyon
Lyon, France
Frederic.Desprez@ens-lyon.fr

Emmanuel Jeannot
LORIA, Université H. Poincaré
Nancy, France
Emmanuel.Jeannot@loria.fr

***Abstract*— The GridRPC model [1] is an emerging standard promoted by the Global Grid Forum (GGF)[1] that defines how to perform remote client-server computation on a distributed architecture. In this model data are sent back to the client at the end of every computation. This implies unnecessary communications when computed data are needed by an other server in further computations.**

Since, communication time is sometimes the dominant cost of remote computation, this cost has to be lowered.

Several tools instantiate the GridRPC model such as NetSolve which is a NES environment developed at the University of Tennessee, Knoxville.

In this paper, we present the modifications we made to the NetSolve protocol in order to overcome this drawback. We have developed a set of new functions and data structures that allow clients to order servers to keep data in place and to redistribute them directly to an other server when needed.

I. Introduction

Due to the progress in networking, computing intensive problems in several areas can now be solved using networked scientific computing. In the same way that the World Wide Web has changed the way that we think about information, we can easily imagine the types of applications we might construct if we had instantaneous access to a supercomputer from our desktop. The GridRPC approach [1] is a good candidate to build Problem Solving Environments on computational Grid. It defines an API and a model to perform remote computation on servers.

Several tools exist that provide this functionality like NetSolve [2], NINF [3], DIET [4], NEOS [5], or RCS [6]. However, none of them do implement data persistence in servers and data redistribution between servers. This means that once a server has finished its computation, output data are immediately sent back to the client and input data are destroyed. Hence, if one of these data is needed for another computation, the client has to bring it back again on the server. This problem as been partially tackled in NetSolve with the request sequencing feature [7]. However, the current request sequencing implementation does not handle multiple servers.

This work is a "*proof of concept*" type of work. We have chosen a GridRPC middleware, implemented persistence and data redistribution features and evaluated the performance of these features on several application kernels. We have chosen NetSolve and modiied its internal communication protocol as well as its API to manage data on the servers. Results show performance improvement on a LAN and on a WAN.

The remaining of this paper is organized as follows. The GridRPC architecture, NetSolve and the objectives of this work are described in Section II. In Section III-A, we describe the modifications made to the server. Section III-B is dedicated to a new data structure that tells how to redistribute objects from a server to an other server. In Section III-C, we describe all the NetSolve client functions involved in data redistribution and data persistence operations. In Section III-D, the modifications done to NetSolve's scheduler are described and a program example is presented in Section IV. Experimental results are given in Section V. Finally and before some concluding remarks, we discuss a generalization of this work in section VI.

II. Background

A. GridRPC Overview

The GridRPC model defines an architecture for executing computations on remote servers. This architecture is composed of three components:

- *the agent* is the manager of the architecture. It knows the state of the system. Its main role is to find servers that will be able to solve as efficiently as possible client requests,
- *servers* are computational resources. Each server registers to an agent and then waits for client requests. Computational capabilities of a server are known as problems (matrix multiplication, sort, linear systems solving, . . .). A server can be sequential (executing sequential routines) or parallel (executing operations in parallel on several nodes),
- *a client* is a program that requests for computational resources. It asks the agent to find a set of servers that will be able to solve its problem. Data transmitted between a client and a server is called object. Thus, an input object is a parameter of a problem and an output object is a result of a problem.

The GridRPC architecture works as follows. First, an agent is launched. Then, servers register to the agent by sending information of problems they are able to solve as well as

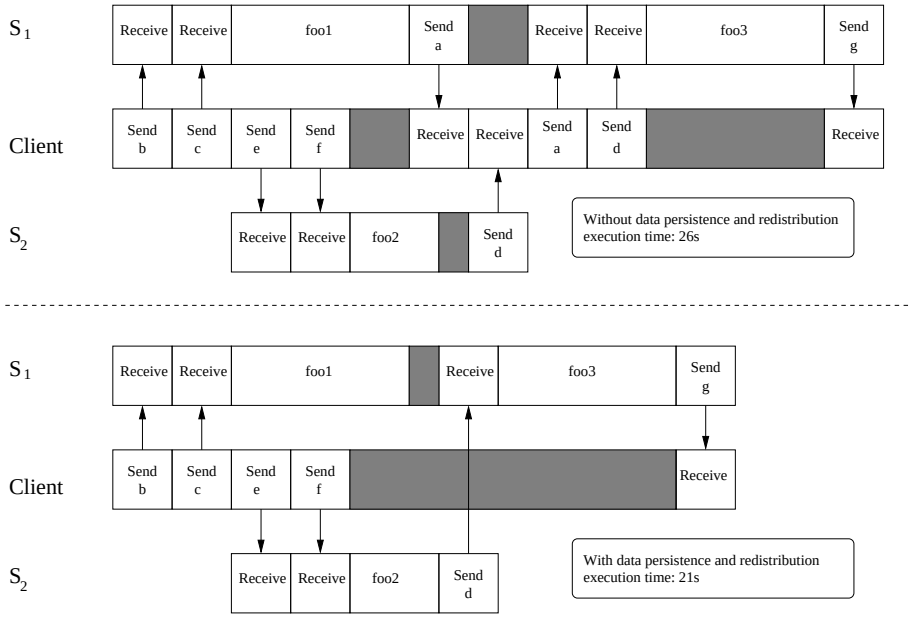
[1] http://www.ggf.org

```
a = foo1(b,c)
d = foo2(e,f)
g = foo3(a,d)
```

(a) Sample C code

Function	Server 1	Server 2
foo1	6s	9s
foo2	2s	3s
foo3	6s	11s

(b) Execution time



(c) Execution without (top) and with (bottom) persistence

Fig. 1. Sample example where data persistence and redistribution is better than retrieving data to the client.

information of the machine on which they are running and the network's speed (latency and bandwidth) between the server and the agent. A client asks the agent to solve a problem. The agent scheduler selects a set of servers that are able to solve this problem and sends back the list to the client. The client sends the input objects to one of the servers. The server performs the computations and returns the output objects to the client. Finally local server objects are destroyed.

This architecture is based on the client-server programmation paradigm. This paradigm is different than other ones such as parallel/distributed programmation. In a parallel program (written in PVM or MPI for instance) data persistence is performed implicitly : once a node has received some data, this data is supposed to be available on this node as long as the application is running (unless explicitly deleted). Therefore, in a parallel program data can be used for several steps of the parallel algorithm. On the over hand, in a GridRPC architecture (the one we address in this paper) no data management is performed. Like in the standard RPC model, request parameters are sent back and forth between the client and the server. A data is not supposed to be available on a server that used it for another step of the algorithm (an new RPC) once a step is finished (a previous RPC has returned). This drawback can lead to very costly computation as the execution and the communications can be performed over the internet.

We show here an example where using data persistence and redistribution is better than retrieving data from the client. Assume a client asks to execute the three functions/problems shown in the sample code Figure 1(a).

Let us consider that the underlying network between the client and the server has a bandwidth of 100 Mbit/s (12.5 Mbytes per seconds). Figure 1(b) gives the execution time for each function and for each server. Finally let us suppose that each object has a size of 25 Mbytes. The GridRPC architecture will execute *foo1* and *foo3* on server S_1 and *foo2* on S_2 and sends the objects in the following order: *b*,*c*,*e*,*f* (Figure 1(c)). Due to the bandwidth, *foo1* will start 4 seconds after the request and *foo2* after 8 seconds. Without data persistence and redistribution *a* will be available on S_1 16 seconds after the beginning and *d* 18 seconds after the beginning (S_2 has to wait that the client has completely received *a* before starting to send *d*). Therefore, after the execution of *foo3*, *g* will be available on the client 26 seconds after the beginning. With data persistence and redistribution, S_2 sends *d* to S_1 which is

available 13 seconds after the beginning of the request. Hence, *g* will be available on the client 21 seconds after the beginning of the request which leads to a 19% improvement.

B. NetSolve and Request Sequencing

NetSolve is tool built at the University of Tennessee and instantiate the GridRPC model [2]. In order to tackle the problem of sending to much data on the Network, the *request sequencing* feature has been proposed since NetSolve 1.3 [7]. Request sequencing consists in scheduling a sequence of NetSolve calls on one server. This is a high level functionality since only two new sequence delimiters `netsl_sequence_begin` and `netsl_sequence_start` are added in the client API. The calls between those delimiters are evaluated at the same time and the data movements due to dependencies are optimized.

However request sequencing has the following deficiencies. First it does not handle multiple servers because no redistribution is possible between servers. An overhead is added for scheduling NetSolve requests. `for` loops are forbidden within sequences, and finally the execution graph must be known at compile time and cannot depend on results computed within the sequence.

Data redistribution is not implemented in the NetSolve's request sequencing feature. This can lead to sub-optimal utilization of the computational resources when, within a sequence, two or more problems can be solved in parallel on two different servers. This is the case, for instance, if the request is composed of the problems *foo1, foo2 and foo3* given Figure 1(c). The performance can be increased if *foo1* and *foo2* can be executed in parallel on two different servers.

C. Goal of our Work

In this work we modified NetSolve and added the data persistence and data redistribution.

Data persistence consists in allowing servers to keep objects in place to be able to use these objects again for a new call without sending them back and forth to and from the client. Data redistribution enables inter-server communications to avoid object moving though the client.

Our modification is backward compatible. Data persistence and data redistribution require the client API to be modified but we want standard clients to continue to execute normally. Moreover, our modifications are standalone. This means that we do not want to use an other software to implement our optimizations. Hence, NetSolve users do not have to download and compile new tools. Finally, our implementation is very flexible without the restrictions imposed by NetSolve's request sequencing feature.

III. Modifications Done to NetSolve

A. Server Modifications

NetSolve communications are implemented using sockets. In this section, we give details about the low level protocols that enable data persistence and data redistribution between servers.

1) Data Persistence: When a server has finished its computations, it keeps all the objects locally, listen to a socket and waits for new orders from the client. So far, the server can receive five different orders.

1) *Exit*. When this order is received, the server terminates the transaction with the client, exits, and therefore data are lost. Saying that the server exits is not completely correct. Indeed, when a problem is solved by a server, a process is forked, and the computations are performed by the forked process. Data persistence is also done by the forked process. In the following, when we say that the server is terminated, it means that the forked process exits. The NetSolve server is still running and it can solve new problems.
2) *Send one input object*. The server must send an input object to the client or to an other server. Once this order is executed, data are not lost and the server is waiting for new orders.
3) *Send one output object*. This order works the same way than the previous one but a result is sent.
4) *Send all input objects*. It is the same as "send one input object" but all the input objects are sent.
5) *Send all output objects*. It is the same as "send one output object" but all the results are sent.

2) Data Redistribution: When a server has to solve a new problem, it has first to receive a set of input objects. These objects can be received from the client or from an other server. Before an input object is received, the client tells the server if this object will come from a server or from the client. If the object comes from the client, the server has just to receive the object. However, if the object comes from an other server, a new protocol is needed. Let call S_1 the server that has to send the data, S_2 the server that is waiting for the data, C and the client.

1) S_2 opens a socket s on an available port p.
2) S_2 sends this port to C.
3) S_2 waits for the object on socket s.
4) C orders S_1 to send one object (input or output). It sends the object number, forward the number of the port p to S_1 and sends the hostname of S_2.
5) S_1 connects to the socket s at port p of S_2.
6) S_1 sends the object directly to S_2 on this socket: data do not go through the client.

B. Client Modifications

1) New structure for the client API: When a client needs a data to stay on a server, three informations are needed to identify this data. (1) Is this an input or an output object? (2) On which server can it be currently found? (3) What is the number of this object on the server?

We have implemented the `ObjectLocation` structure to describe these needed informations. `ObjectLocation` has 3 fields:

1) `request_id` which is the request number of the non-blocking call that involves the requested data. The request id is returned by the `netslnb` standard NetSolve

function, that performs a non blocking remote execution of a problem. If `request_id` equals -1, this means that the data is available on the client.
2) `type` can have two values: `INPUT_OBJECT` or `OUTPUT_OBJECT`. It describes if the requested object is an input object or a result.
3) `object_number` is the number of the object as described in the problem descriptor.

2) Modification of the NetSolve code: When a client asks for a problem to be solved, an array of `ObjectLocation` data structures is tested. If this array is not `NULL`, this means that some data redistribution have to be issued. Each element of the array corresponds to an input object. For each input object of the problem, we check the `request_id` field. If it is smaller than 0, no redistribution is issued, everything works like in the standard version of Netsolve. If the `request_id` field is greater than or equal to zero then data redistribution is issued between the server corresponding to this request (it must have the data), and the server that have to solve the new problem.

C. Set of New Functions

In this section, we present the modifications of the client API that uses the low-level server protocol modifications described above. These new features are backward compatible with the old version. This means that an old NetSolve client will have the same behavior with this enhanced version: all the old functions have the same semantic, except that when doing a non-blocking call, data stay on the server until a command that terminates the server is issued. These functions have been implemented for both C and Fortran clients. These functions are very general and can handle various situations. Hence, unlike request sequencing, no restriction is imposed to the input program. In section IV, a code example is given that uses a subset of these functions.

1) Wait Functions: We have modified or implemented three functions: `netslwt`, `netslwtcnt` and `netslwtnr`. These functions block until computations are finished. With `netslwt`, the data are retrieved and the server exits. With `netslwtcnt` and `netslwtnr`, the server does not terminate and other data redistribution orders can be issued. The difference between these two functions is that unlike `netslwtcnt`, `netslwtnr` does not retrieve the data.

2) Terminating a Server: The `netslterm` orders the server to exit. The server must have finished its computation, local object are then lost.

3) Probing Servers: As in the standard NetSolve, `netslpr` probes the server. If the server has finished its computations, results are not retrieved and data redistribution orders can be issued.

4) Retrieving Data: A data can be retrieved with the `netslretrieve` function. Parameters of this functions are the type of the object (input or output), the request, the object number and a pointer where to store the data.

5) Redistribution Function: `netslnbdist`, is the function that performs the data redistribution. It works like the standard non-blocking call `netslnb` with one more parameter: an `ObjectLocation` array, that describes which objects are redistributed and where they can be found.

D. Agent Scheduler Modifications

The scheduling algorithm used by NetSolve is Minimum Completion Time (MCT) [8] which is described Figure 2. Each time a client send a request MCT chooses the server that minimizes the execution time of the request assuming no major change in the system state.

1 **For all** server S that can resolve the problem
2 $D_1(S)$ = estimated amount of time to transfer input and output data.
3 $D_2(S)$ = estimated amount of time to solve the problem.
4 Choose the server that minimizes $D_1(S) + D_2(S)$.

Fig. 2. *MCT algorithm*

We have modified the agent's scheduler to take into account the new data persistence. The standard scheduler assumes that all data are located on the client. Hence, communication costs do not depend on the fact that a data can already be distributed. We have modified the agent's scheduler and the protocol between the agent and the client in the following way. When a client asks the agent for a server, it also sends the location of the data. Hence, when the agent computes the communication cost of a request for a given server, this cost can be reduced by the fraction of data already hold by the server.

IV. Code Example

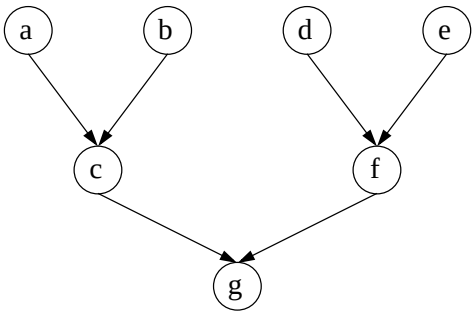
In figure 3 we show a code that illustrates the features described in this paper. It executes 3 matrix multiplications: `c=a*b`, `d=e*f`, and `g=d*a` using the blas `dgemm` function provided but Netsolve, where `a` is redistributed from the first server and `d` is redistributed from the second one. We will suppose that matrices are correctly initialized and allocated. In order to simplify this example we will suppose that each matrix has `n` rows and columns and tests of requests are not shown.

In the two `netslnb` calls different parameters of `dgemm` ($c = \beta \times c + \alpha \times a \times b$, for the first call) are passed such has the matrix dimension (always `n` here), the need to transpose input matrices (not here), the value of α and β (respectively 1 and 0) and pointers to input and output objects. All these objects are persistent and therefore stay on the server: they do not move back to the client.

Then the redistribution is computed. An array of `ObjectLocation` is build and filled for the two objects that need to be redistributed (`a` and `d`).

```
ObjectLocation *redist;

netslmajor("Row");
trans="N";
alpha=1;
beta=0;

/* c=a*b */
request_c=netslnb("dgemm()",&trans,&trans,n,n,n,&alpha,a,n,b,n,&beta,c,n);
/* after this call c is only on the server*/

/* d=e*f */
request_d=netslnb("dgemm()",&trans,&trans,n,n,n,&alpha,e,n,f,n,&beta,d,n);
/* after this call d is only on the server*/

/* COMPUTING REDISTRIBUTION */
/* 7 input objects for dgemm */
nb_objects=7;
redist=(ObjectLocation*)malloc(nb.objects*sizeof(ObjectLocation));

/* All objects are first supposed to be hosted on the client */
for(i=0;i<nb_object;i++)
redist[i].request.id=-1;

/* We want to compute g=d*a */

/* a is the input object No 4 of dgemm and the input object No 3 of request_c */
redist[4].request_id=request_c;
redist[4].type=INPUT_OBJECT;
redist[4].object_number=3;

/* d is the input object No 3 of dgemm and the output object No 0 of request_d */
redist[3].request_id=request_d;
redist[3].type=OUTPUT_OBJECT;
redist[3].object_number=0;

/*g=d*a*/
request_g=netslnbdist("dgemm()",redist,&trans,&trans,n,n,n,&alpha,NULL,n,NULL,n,
                      &beta,g,n);

/*wait for g to be computed and retrieve it*/
netslwt(request_g);

/*retrieve c*/
netslretrieve(request_c,OUTPUT_OBJECT,0,c);

/*Terminate the server that computed d*/
netslterm(request_d);
```

Fig. 3. code example




```
c=a*b
f=d*e
g=c*f
```

Fig. 4. Matrix multiplications program task graph.

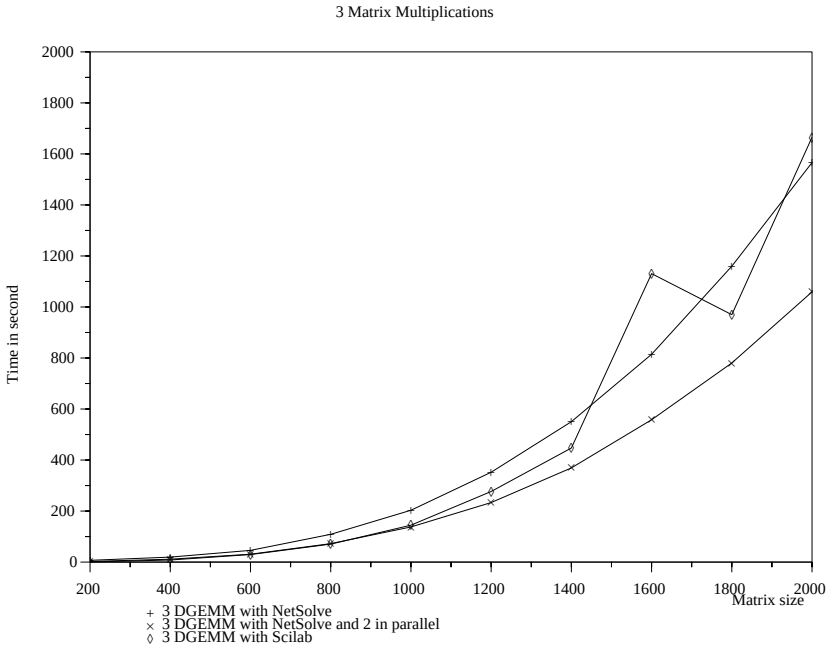


Fig. 5. Matrix multiplications using NetSolve on a cluster of PCs.

The call to `netslnbdist` is similar to previous netslnb call except that the redistribution parameter is passed.

At the end of the computation, a *wait* call is performed for the computation of `g`, the matrix `c` is retrieved and the server that computed `d` is terminated.

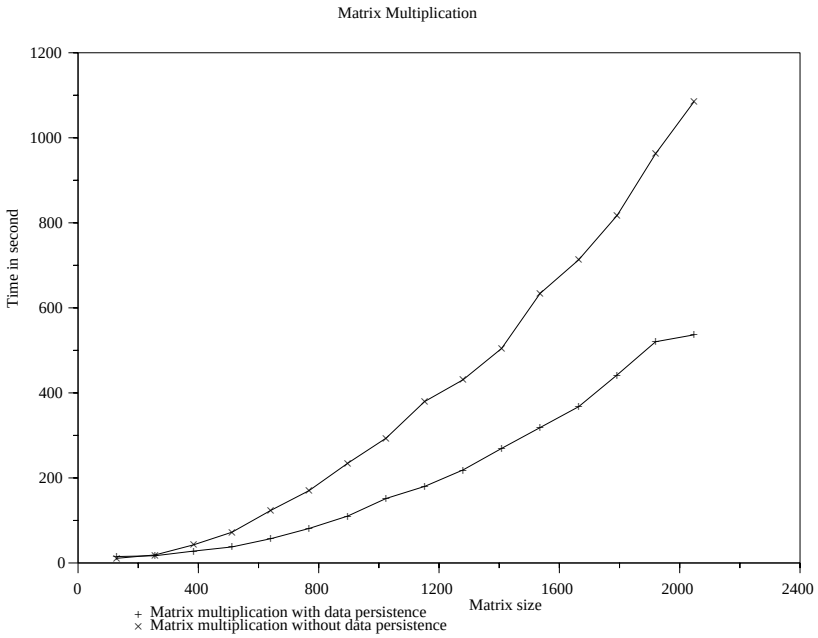
V. Experiments

Scilab is a tool heavily used in the mathematic community [6]. As Matlab, it allows to execute scripts for engineering and scientific computations. However, it has some limitations since it is not parallelized. The goal of Scilab$_{//}$ [9], developed in the OURAGAN project[2] is to allow an efficient and transparent execution of Scilab in a grid environment. Various approaches have been implemented in order to meet these objectives. One of these is to execute Scilab computations on dedicated servers distributed over the Internet. In order to achieve this goal, we chose to use NetSolve [10], [2] as a middleware between the Scilab console and our computational servers.

Figures 5 and 6 show our experimental results using NetSolve as a NES environment for solving matrix multiplication problems in a grid environment.

In Figure 5, we ran a NetSolve client that performs 3 matrix multiplications using 2 servers. The client, agent, and servers are in the same LAN and are connected through Ethernet. Computations and task graphs are shown in Figure 4. The first two matrix multiplications are independent and can be done in parallel on two different servers. We see that the time taken by Scilab is about the same than the time taken using NetSolve when sequentializing the three matrix multiplications. When doing the first two ones in parallel on two servers using the redistribution feature, we see that we gain exactly one third of the time, which is the best possible gain. These results show that NetSolve is very efficient in distributing matrices in a

[2] `http://graal.ens-lyon.fr/~desprez/OURAGAN`



1) $C_{11} = A_{11}B_{11}$
2) $C_{22} = A_{21}B_{12}$
3) $C_{12} = A_{11}B_{12}$
4) $C_{21} = A_{21}B_{11}$
5) $C_{11} = C_{11} + A_{12}B_{21}$
6) $C_{22} = C_{22} + A_{22}B_{22}$
7) $C_{12} = C_{12} + A_{12}B_{22}$
8) $C_{21} = C_{21} + A_{22}B_{21}$

Fig. 6. Matrix multiplication using block decomposition.

LAN and that non-blocking calls to servers are helpful for exploiting coarse grain parallelism.

Then, we have performed a matrix multiplication (Figure 6) The client and agent were located in one University (Bordeaux) but servers were running on the nodes of a cluster located in Grenoble [3]. The computation decomposition done by the client is shown in Figure 6. Each matrix is decomposed in 4 blocks, each block of matrix A is multiplied by a block of matrix B and contributes to a block of matrix C. The first two matrix multiplications were performed in parallel. Then, input data were redistributed to perform matrix multiplications 3 and 4.The last 4 matrix multiplications and additions can be executed using one call to the level 3 BLAS routine DGEMM and requires input and output objects to be redistributed. Hence, this experiment uses all the features we have developed. We see that with data persistence (input data and output data are redistributed between the servers and do not go back to the client), the time taken to perform the computation is more than twice faster than the time taken to perform the computation without data persistence (in that case, the blocks of A, B, and C are sent back and forth to the client). This experiment demonstrates how useful the data persistence and redistribution features that we have implemented within NetSolve are.

VI. Generalization

Results in the previous section show that data persistence and redistribution improve performance of NES. Therefore data management appears to be an important feature for GridRPC environment. In order the client to be able to manage data three functionalities are required:

1) *Data handler*. The handler must describe the state of data (persistent or not) and its location. In this paper the `ObjectLocation` structure play the role.
2) *Storage on the server*. In order to implement data persistence each server must be able to temporary store data. Complex memory management such as dumping data to disk might be required for some class of applications.
3) *Data operations*. Given a data handler, one must be able to destroy this data on a remote server, redistribute this data from one server to an other, or retrieve this data to the client. More complex functionalities such as managing duplication might also be wanted. Moreover, some operations on how to manage consistency (what to do if the same data is duplicated on more than one server) have to be implemented. An other issue to address concern the time limitation of the persistence. It is not reasonable to assume that any data can be persistent forever. When too much data is stored in the memory of a server, mechanism close to processor cache algorithms have to be imagine. For instance data can be dumped to disk or moved to persistent remote storage facilities when memory is required. Data can also be deleted after a given amount of time depending on its type and some specifications given by the client.

Discussions have started within the GridRPC Working Group of the GGF to take this feature into account.

[3]Grenoble and Bordeaux are two french cities separated by about 500 miles

VII. Conclusion and future work

The GridRPC model is an emerging standard for NES middleware. However, this standard does not deal with data management. This leads to sub-optimal communication cost when requests require intermediate objects to be computed on distant servers.

Data persistence (the ability of a server to keep data locally) and data redistribution (the ability of a server to send its data to an other server) are studied here. This paper is a *proof of concept* work, where we show how we added data management to NetSolve, a GridRPC NES.

Our contributions are the following. We have modified the server in order to be able to keep data in place on servers after computation. The server waits for orders from the client and is able to redistribute data to an other server when needed. We have modified the client API in order to simply write client programs involving data persistence and data redistribution. Then, we have modified the agent's scheduler. Allocation decisions take into account the fact that some data may already be distributed. These modifications keep the backward compatibility with old Netsolve clients. Finally, these features are standalone and there is no need to download or compile an other software.

Results show a real improvement of the NetSolve environment thanks to data persistence and redistribution either on LAN or on WAN.

Future works are directed towards the following directions. First, we want to implement new functionalities such as deleting some data on a server. Second, we need to enhance the scheduler to be able to take more accurate decisions. Third, we are aware that using data persistence and data redistribution requires to rewrite NetSolve client programs. We would like to simplify the client API in order to increase the transparency of the proposed features. Finally, we want to implement data redistribution and data persistence for parallel servers. This last development requires the development of a data redistribution routine between servers that will be able to transfer huge distributed data sets.

Acknowledgment

The authors would like to thank Eddy Caron from LIP, ENS-Lyon for fruitful discussions on data redistribution and its generalization.

References

[1] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova, "A GridRPC Model and API for End-User Applications," Dec. 2003, https://forge.gridforum.org/projects/gridrpc-wg/document/GridRPC_EndUse%r_16dec03/en/1.

[2] H. Casanova and J. Dongarra, "NetSolve: A Network-Enabled Server for Solving Computational Science Problems," *International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, no. 3, pp. 212 – 213, Fall 1997.

[3] S. S. Hidemoto Nakada, Mitsuhisa Sato, "Design and implementations of ninf: towards a global computing infrastructure," *Future Generation Computing Systems, Metacomputing Issue*, vol. 15, pp. 649–658, 1999.

[4] "DIET," `http://graal.ens-lyon.fr/DIET/`.

[5] "NEOS," `http://www-neos.mcs.anl.gov/`.

[6] P. Arbenz, W. Gander, and J. Moré, "The remote computational system," *Parallel Computing*, vol. 23, no. 10, pp. 1421–1428, 1997.

[7] D. C. Arnold, D. Bachmann, and J. Dongarra, "Request sequencing: Optimizing communication for the grid," in *Euro-Par 2000 Parallel Processing, 6th International Euro-Par Conference*, vol. volume 1900 of Lecture Notes in Computer Science. Munich Germany: Springer Verlag, Aug. 2000, pp. 1213?–1222.

[8] M. Maheswaran, S. Ali, H. J. Siegel, D. Hengsen, and R. F. Freund, "Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing system," in *Proceedings of the 8th Heterogeneous Computing Workshop (HCW '99)*, april 1999.

[9] E. Caron, S. Chaumette, S. Contassot-Vivier, F. Desprez, E. Fleury, C. Gomez, M. Goursat, E. Jeannot, D. Lazure, F. Lombard, J. Nicod, L. Philippe, M. Quinson, P. Ramet, J. Roman, F. Rubi, S. Steer, F. Suter, and G. Utard, "Scilab to Scilab$_{//}$, the OURAGAN Project," *Parallel Computing*, vol. 27, no. 11, 2001.

[10] D. C. Arnold and J. Dongarra, "The NetSolve Environment: Progressing Towards the Seamless Grid," in *International Conference on Parallel Processing (ICPP-2000)*, Toronto Canada, Aug. 2000.