Scientific Programming 13 (2005) 333–354
IOS Press

Managing data persistence in network enabled servers[1]

Eddy Caron[a,*], Bruno DelFabbro[b], Frédéric Desprez[a], Emmanuel Jeannot[c] and Jean-Marc Nicod[b]
[a]*GRAAL Project, LIP ENS Lyon, 46 Alle d'Italie, 69364 Lyon Cedex 07, France*
E-mail: Eddy.Caron@ens-lyon.fr
[b]*GRAAL Project, LIFC, Université de Franche-Comté, 16 route de Gray, 25030 Besançon Cedex, France*
E-mail: delfabbro@lifc.univ-fcomte.fr
[c]*ALGORILLE Project, LORIA,INRIA-Lorraine, Nancy, France*
E-mail: Emmanuel.Jeannot@loria.fr

Abstract. The GridRPC model [17] is an emerging standard promoted by the Global Grid Forum (GGF) that defines how to perform remote client-server computations 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 computations, this cost has to be lowered. Several tools instantiate the GridRPC model such as NetSolve developed at the University of Tennessee, Knoxville, USA, and DIET developed at LIP laboratory, ENS Lyon, France. They are usually called Network Enabled Servers (NES). In this paper, we present a discussion of the data management solutions chosen for these two NES (NetSolve and DIET) as well as experimental results.

1. Introduction

Due to the progress in networking, computing intensive problems from several areas can now be solved using network 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 kind of applications we might construct if we had instantaneous access to a supercomputer from our desktop. The GridRPC approach [20] 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. In such a paradigm, a client can submit a request for solving a problem to an agent that chooses the best server amongst a set of candidates. The choice is made from static and dynamic information about software and hardware resources. Request can be then processed by sequential or parallel servers. This paradigm is close to the RPC (*Remote Procedure Call*) model. The GridRPC API is the Grid form of the classical Unix RPC approach. They are commonly called Network Enabled Server (NES) environments [16].

Several tools exist that provide this functionality like NetSolve [7], Ninf [13], DIET [4], NEOS [18], or RCS [1]. However, none of them do implement a general approach for data persistence 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 [2]. However, the current request sequencing implementation does not handle multiple servers.

In this paper, we present how data persistence can be handled in NES environments. We take two existing environments (NetSolve and DIET) and describe how

[1]This work was supported in part by the ACI GRID (ASP) and the RNTL (GASP) from the French ministry of research.
*Corresponding author.

ISSN 1058-9244/05/$17.00

we implemented data management in their kernels. For NetSolve, it requires to change the internal protocol, the client API and the request scheduling algorithm. For DIET we introduce a new service, called the Data Tree Manager (DTM), that identify and manage data within this middleware. We evaluate the gain that can be obtained from these features on a grid. Since we show that data management can greatly improve application performance we discuss a standardization proposal.

The remaining of this paper is organized as follows. In Section 2, we give an overview of Network Enabled Server (NES) architecture. We focus on NetSolve and DIET. We show why this is important to enable data persistence and redistribution to NES. We describe how we implemented data management in NetSolve and DIET respectively in Section 3 and in Section 4. Experimental results are presented in Section 5. In Section 6 we discuss the standardization of data management in NES. Finally, Section 7 concludes the paper.

2. Background

2.1. Network enabled server architectures

2.1.1. General architecture

The NES model defines an architecture for executing computation 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, etc.). 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 NES 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 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 computation and returns the output objects to the client. Finally local server objects are destroyed.

This client API for such an approach has been standardized within the Global Grid Forum. The GridRPC working group [12] proposed an API that is instantiated by several middleware such as DIET, Ninf, NetSolve, and XtremWeb.

2.1.2. NetSolve

NetSolve [7] (Fig. 1) is a tool built at the University of Tennessee and instantiate the GridRPC model described above. It is out of the scope of this paper to completely describe NetSolve in detail. In this section we focus only on data management.

2.1.2.1. Request sequencing

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 [2]. 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 to schedule NetSolve requests. Indeed, the whole Directed Acyclic Graph of all the NetSolve calls within the sequence is built before being sent to the chosen computational server. Second, `for` loops are forbidden within sequences, and finally the execution graph must be static 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 Fig. 4. The performance can be increased if *foo1* and *foo2* can be executed in parallel on two different servers.

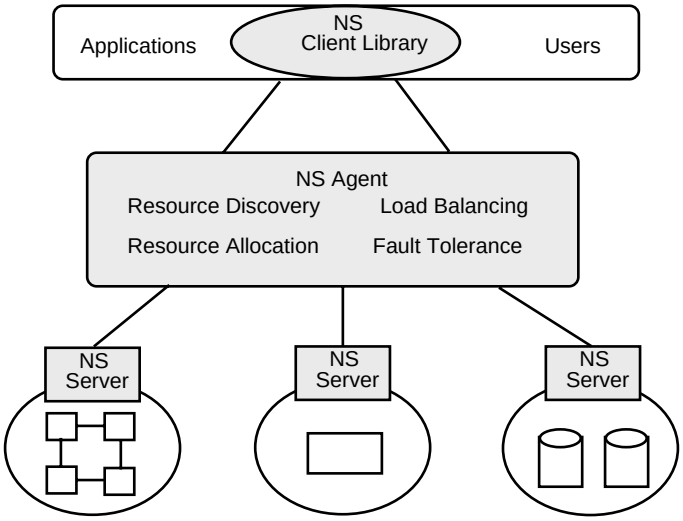


Fig. 1. NetSolve architecture.

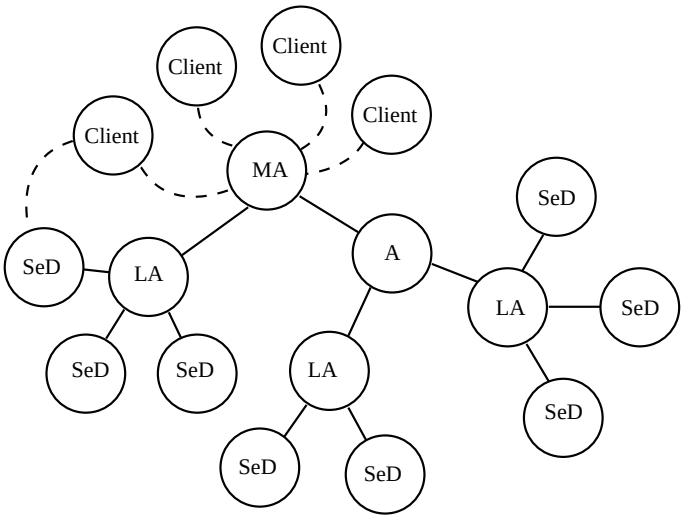


Fig. 2. DIET Architecture.

2.1.2.2. Distributed storage infrastructure

To make a data persistent and to take advantage of its placement in the infrastructure, NetSolve proposes the Distributed Storage Infrastructure. The DSI helps the user for controlling the placement of data that will be accessed by a server (see Fig. 3). Instead of multiple transmissions of the same data, DSI allows the transfer of the data once from the client to a storage server. Considering these storage servers closer from computational servers than from the client, the cost of transferring data will be cheaper. NetSolve is able to manage several DSI. Currently, NetSolve proposes this storage service using IBP (Internet Backplane Protocol).[2] Files or items managed by a DSI are called DSI objects. To generate a DSI data, the client has to know the server in which it wants to store its data. Note that the data location is not a criteria for the choice of a computational server. NetSolve maintains its own File Allocation Table to manage DSI objects. Typically, when a request is submitted to a NetSolve Server, the server looks for input data and verify its existence in its FAT. If the data is referenced (the client had passed a DSI object), data

[2] http://loci.cs.utk.edu/.

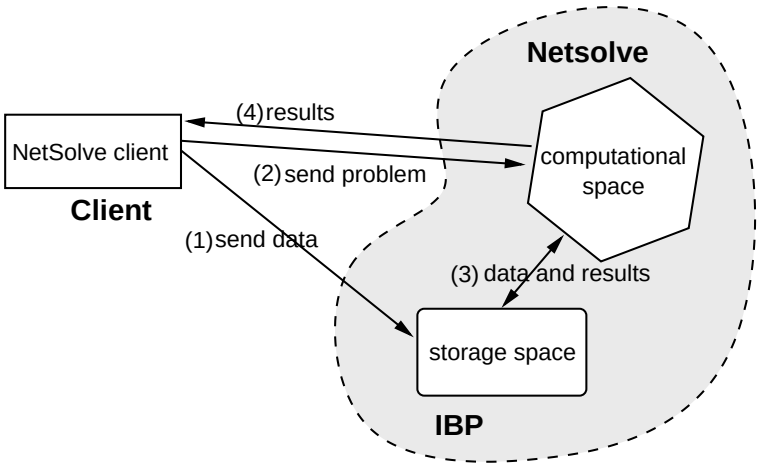


Fig. 3. Distributed storage infrastructure.

is get from the storage server, the server gets it from the client elsewhere.

DSI improves the data transfer but does not prevent from data going back and forth from computational servers to storage servers. Indeed, this feature does not fully implement data persistence and therefore may lead to over-utilization of the network.

2.1.3. DIET architecture

NetSolve and Ninf projects are built on the same approach. Unfortunately, in these environments, it is possible to launch only one agent responsible of the scheduling for a given group of computational servers. [3] The drawback of the mono-agent approach is that the agent can become bottleneck if a large number of requests have to be processed at the same time. Hence, NetSolve or Ninf cannot be deployed for large groups of servers or clients.

In order to solve this problem, DIET proposes to distributed the load of the agent work. It is replaced by several agents which organization follows two approaches: a peer-to-peer multi-agents approach that helps system robustness [6] and a hierarchical approach that helps scheduling efficiency [9]. This repartition offers two main advantages: first, we assume a better load balancing between the agents and a higher system stability (if one of the agents dies, a reorganization of the others is possible to replace it). Then, it is easier to manage each group of servers and agents by delegation which is useful for scalability. DIET is built upon several components:

[3] In Ninf, a multi-agents platform exists (Metaserver) but each agent has the global knowledge of the entire platform.

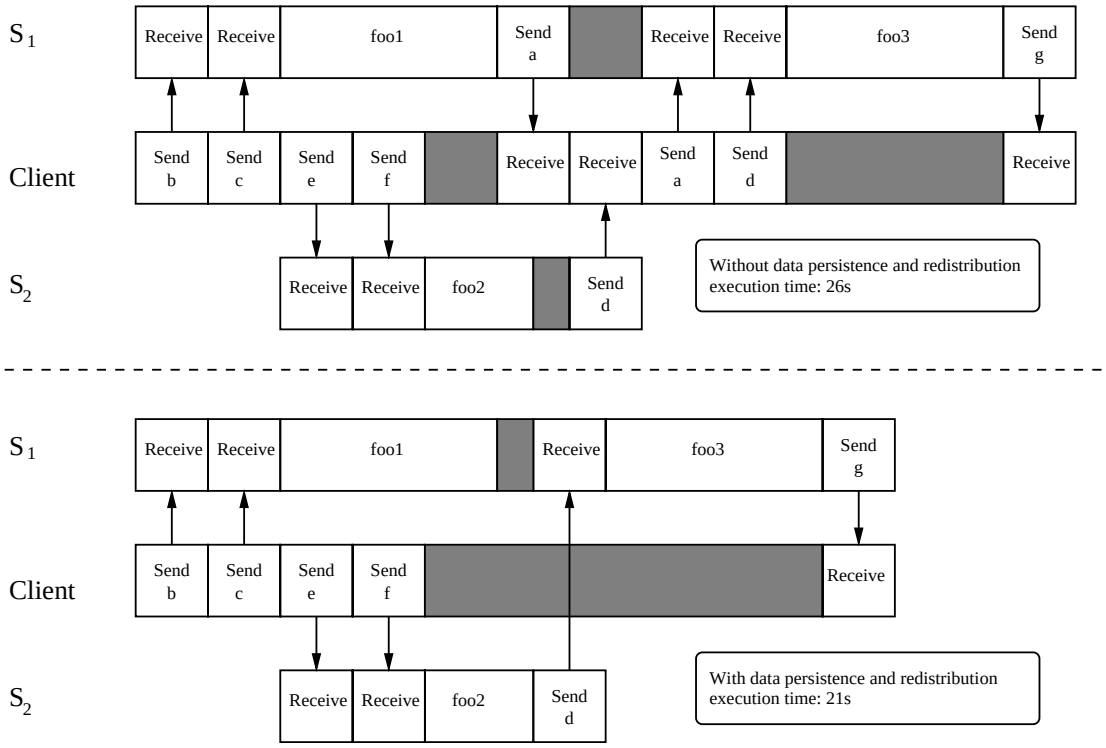
– a *client* is an application that uses DIET to solve problems. Several client types must be able to connect to DIET. A problem can be submitted from a Web page, a problem solving environment such as Scilab [3] or Matlab or from a compiled program.
– a *Master Agent* (*MA*) is directly linked to the clients. It is the entry point of our environment and thus receives computation requests from clients attached to it. These requests refer to some DIET problems that can be solved by registered servers. Then the MA collects computation abilities from the servers and chooses the best one. A MA has the same information than a LA, but it has a global and high level view of all the problems that can be solved and of all the data that are distributed in all its subtrees.
– a *Leader Agent* (*LA*) forms a hierarchical level in DIET. It may be the link between a Master Agent and a SeD or between two Leader Agents or between a Leader Agent and a SeD. It aims at transmitting requests and information between Agents and several servers. It maintains a list of current requests and the number of servers that can solve a given problem and information about the data distributed in its subtrees.
– a *Server Daemon* (*SeD*) is the entry point of a computational resource. The information stored on an SeD is a list of the data available on its server (with their distribution and the way to access them), the list of problems that can be solved on it, and all information concerning its load (memory available, number of resources available, ...). A SeD declares the problems it can solve to its parent. For instance, a SeD can be located on the entry point of a parallel computer.

```
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. 4. Sample example where data persistence and redistribution is better than retrieving data to the client.

A new DIET client contacts a Master Agent (the closest for instance) and posts its request. The Master Agent transmits the request to its subtrees[4] to find data already present in the platform and servers that are able to solve the problem. The LAs which receive the request forward it down to every one of their sub-trees which contains a server that might be involved in the computation and wait for the responses. The requests traverse the entire hierarchy down to the SeDs. When a SeD receives a request, it sends a response structure to its father. It fills the fields for the variables it owns, leaving a null value for the others. If it can solve the problem, it also puts an entry with its evaluated computation time acquired from our performance forecasting tool FAST [19]. Each LA gathers responses coming from its children and aggregates them into a structure.

The scheduling operations are realized at each level of the tree when the response is sent back to the Master Agent. Note that a time-out is set and when an agent has not got a response over a given time, this response is ignored. However, this time-out is not an informa-

[4] An extension is possible for the multi-agent approach: broadcast the request to the others MA considering them as Leader Agents.

tion enough to say that an agent has failed. When the responses come back to the MA, it is able to take a scheduling decision. The evaluated computation and communication times are used to find the server with the lowest response time to perform the computation. Then the MA is able to send the chosen server reference to the client (it is also possible to send a bounded list of best servers to the client). Then, the Master Agent orders the data transfer. Here we can distinguish two cases: data resides in the client and are transferred from the client to the chosen server or data are already inside the platform and are transferred from the servers that holds them to the chosen server. Note that these two operations can be processed in a parallel way. Once data are received by the server, computation can be done. The results may be sent to the client. For performance issues, data are let in the last computational server if possible.

2.2. On the importance of data management in NES

A GridRPC environment such as NetSolve and DIET is based on the client-server programming paradigm. This paradigm is different than other ones such as parallel/distributed programming. 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.

However, in a GridRPC architecture 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 high execution time as the execution and the communications can be performed over the Internet.

2.2.1. Motivating example

Now we give an example where the use of data persistence and redistribution improves the execution of a GridRPC session. Assume that a client asks to execute the three functions/problems shown in the sample code given in Fig. 4(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 4(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* (Fig. 4). Due to the bandwidth limitation, *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 of the session 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.

2.2.2. Goal of the work

In this paper, we show how to add data management into NES environments. We added data persistence and data redistribution to NetSolve and DIET and therefore modified the client API.

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 from and to the client. Data redistribution enables inter-server communications to avoid object moving though the client.

Our modifications to NetSolve are backward compatible. Data persistence and data redistribution require the client API to be modified but standard client programs continue to execute normally. Moreover, our modifications are stand-alone. This means that we do not 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.

We also proposed a model of distributed data management in DIET. The DIET data management model is based on two key elements: the data identifiers and the Data Tree Manager (DTM) [10,11]. To avoid multiple transmissions of the same data from a client to a server, the DTM allows to leave data inside the platform after computation while data identifiers will be used further by the client to reference its data.

3. New data management in NetSolve

In this section we describe how we have implemented data redistribution and persistence within NetSolve. This required to change the three components of the software: server, client, and agent.

3.1. 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.

3.1.1. Data persistence

When a server has finished its computation, 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 computation 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.

3.1.2. 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 on 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.

3.2. Client modifications

3.2.1. New Structure for the Client API

When a client needs a data to stay on a server, three information 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 informations needed. `ObjectLocation` has 3 fields:

1. `request id` which is the request number of the non-blocking call that involves the data requested. 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.

3.2.2. 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 has to solve the new problem.

3.2.3. 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 starting 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. They are very general and can handle various situations. Hence, unlike request sequencing, no restriction is imposed to the input program. In Section 3.4, a code example is given that uses a subset of these functions.

3.2.3.1. Wait functions

We have modified or implemented three functions: `netslwt`, `netslwtcnt` and `netslwtnr`. These functions block until the current 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.

3.2.3.2. Terminating a server

The `netslterm` orders the server to exit. The server must have finished its computation. Local object are then lost.

3.2.3.3. Probing servers

As in the standard NetSolve, `netslpr` probes the server. If the server has finished its computation, results are not retrieved and data redistribution orders can be issued.

3.2.3.4. 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.

3.2.3.5. 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.

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. 5. MCT algorithm.

3.3. Agent scheduler modifications

The scheduling algorithm used by NetSolve is Minimum Completion Time (MCT) [15] which is described in Fig. 5. Each time a client sends a request MCT chooses the server that minimizes the execution time of the request assuming no major change in the system state.

We have modified the agent's scheduler to take into account the new data persistence features. 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.

3.4. Code example

In Fig. 6 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 `DGEMM` function of the level 3 BLAS provided by 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 also 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 as the matrix dimension (always `n` here), the need to transpose input matrices (not used 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.

```
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. 6. NetSolve persistence code example.

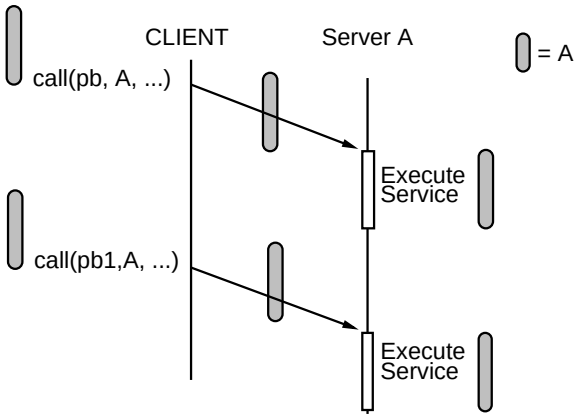


Fig. 7. Sending A twice.

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`). 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.

In Section 5.3, we present our experimental results on executing a set of `DGEMM` requests both on a LAN and on a WAN.

4. Data management in DIET

We have developed a data management service in the DIET platform. Our motivation was based on the need to decrease the global computation time. A way to achieve such a goal is to decrease data transfers between clients and the platform when possible. For example, a client that submits two successive calls with the same input data needs to transfer them twice (see Fig. 7). Our goal is to provide a service that allows only one data transfer as shown in Fig. 8. An other objective is to allows the use of the data already stored inside the platform in later computations and more generally in later sessions or by others clients. This is why data stored needed to be handled by an unique identifier. Our service has also to fit with DIET platform characteristics, and this is why our components are build in a hierarchical way. After a short description of the principles we retain in order to build a data management service in DIET, we review the various components of our implementation called Data Tree Manager [10].

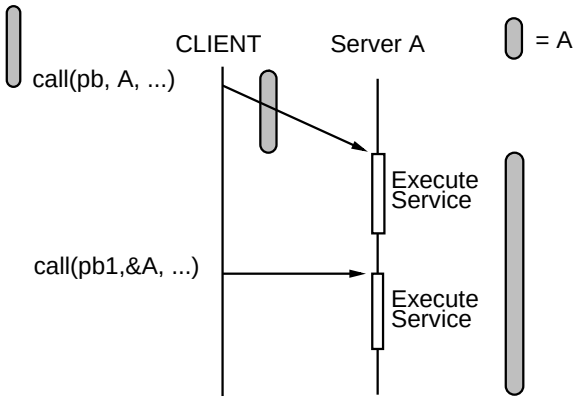


Fig. 8. Sending A only once.

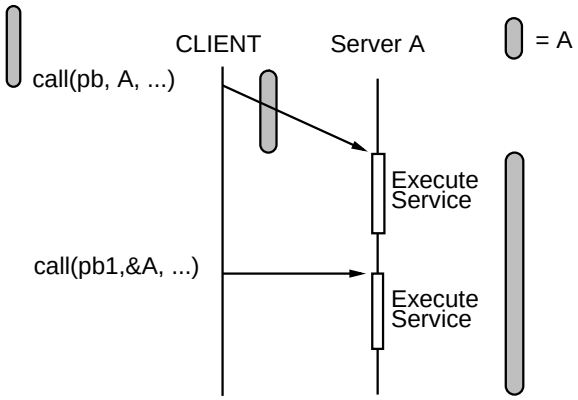


Fig. 9. Two successive calls.

4.1. Principles

In this section, we present the basic functionalities that we choose for a data management service in such an ASP environment.

4.1.1. Data storage

A data can be stored onto a disk or in memory. In NES environments, a challenge is to store data as near as possible to a computational server where they will be needed. In addition, physical limitations of storage resources will imply the definition of a data management policy. Simple algorithms as LRU will be implemented in order to remove the most older data. This will avoid to overload the system.

4.1.2. Data location

When a data item has been sent once from a client to the platform, the data management service has to be able to find where data is stored to use it in other computations on other servers. Furthermore, in order

to obtain a scalable infrastructure, we need to separate the logical view of the data from its physical location. Even if the solution of metadata [8] is elegant, a data management service in NES environments has not exactly the same characteristics than other data management systems implemented in Grid Computing Environments. In fact, in these environments, clients need to access huge data for analysis. Hence, these systems are built in order to provide a reliable access to data that are geographically distributed. In ASP environments, numerical applications to which NES platforms give access have generally data that are produced and directly accessed by the client that sends the request. ASP environments have to give a reliable access to computational servers even if the problematic of data access by clients is also a constraint. This is why it is not necessary to define data along their characteristics. Nevertheless, it is mandatory to fully identify data that are stored inside the platform.

4.1.3. Data movement

As seen above, a data management service in ASP environments is able to store and locate data. But when data is required for more than one computation on more than one server, it is also mandatory to be able to move data between computational servers. In fact, if we consider that time to transfer data between servers is smaller than time to transfer data between clients and servers, we need to define a data movement mechanism. Obviously, when data is moved from one server to an other computational server, information on its location have to be updated.

4.1.4. Persistence mode

A data can be stored inside the platform and moved between storage resources. But, have all data sent by clients or produced by servers to be stored inside the platform? For obvious performance motivations, it is better to limit data persistence to those that are really useful. We think that only clients know which data have to stay inside the system. Hence, this is why we define a persistence mode in the help of which clients can tell if their data should be stored or not.

4.1.5. Security

Once data are stored inside the platform, we need to define a policy to make secured operations on data. In fact, data stored inside the platform can be shared between clients. However, all the clients of the platform are not able to realize all operations on all data. As data stored are identified inside the platform, only the client that has produced the data has to be informed of the identifier that has been bound to its data in order to use it for later computation requests. Moreover, in collaborative projects for example, a client may want to share its stored data with other researchers but he does not want them to delete its data. We propose to add an access key in addition to the identifier. Thus, if a client wants to get read/write rights on a specified data, he has to join this key to the data identifier. Indeed, if the client that has produced the data does not want the others to have write access on it, he just have to provide the identifier. This leaves the responsibility of the management of its own data to the client. Simple mechanisms such as *md5, sha1* algorithms or routines like *urandom* will be chosen to generate such a key.

4.1.6. Fault tolerance

The fault tolerance policy is directly linked to the consistency policy. In fact, our approach does not define fault recovery mechanisms but only a consistency mechanism of the infrastructure when faults occur. Thus, only a context/contents model is defined. We ensure that all operations (add, remove) made on data by clients are made such that all the infrastructure is consistent. If a component that manages the physical data fails (named DataManager), updates on the architecture are made. We distinguish two possible cases of fault. A component that manages the logical view of data fails (named LocManager) or DataManager fails. If a LocManager fails, all its subtrees are considered as lost. We only ensure that the parent of the LocManager removes all references of data referenced on this branch. If a DataManager fails, we ensure that all references of data owns by it are removed in the hierarchy. No data recovery is made. We also consider that all data transfers are realized in a correct way but we make sure that updates are realized only when transfers are complete. A solution will be to replicate data.

4.1.7. Data sources heterogeneity

Generally, a data is sent from the local machine of a client. However, it is also possible that a client does not owns the data it wants to send to the platform but only knows its location. Hence, we propose to give the possibility for a client to inform the server to pull data from a remote storage depot that is extern to the platform. This model has to deal with the support heterogeneity. We have first developed a model that allow the use of ftp and http protocols. These models have to be completed to interact with other protocols such as gridFTP. This approach is quite similar to the Stork approach for multi-protocols data transfers presented in [14].

4.1.8. Replication

One mandatory aspect of a data management service is to provide a data replication policy. In fact, the need of data replication is particularly required for parallel tasks that share data. Thus, a data management service needs to provide an API in order to move or replicate data between computational servers. This API will be used by a task scheduler for example.

4.2. The DIET Data tree manager

The data management service we implemented is based on the principles defined above. In this section, we present our implementation.

4.2.1. The persistence mode

A client can choose whether a data will be persistent inside the platform or not. We call this property the `persistence mode` of a data. We have defined several modes of data persistence as shown in Table 1.

4.2.2. The data identifier

When a data is stored inside the platform, an identifier is assigned to it. This identifier (also known as data handler) allows us to point out a data in an unique way within the architecture. It is clear that a client has to know this identifier in order to use the corresponding data. Currently, a client knows only the identifiers of the persistent data it has generated. It is responsible for propagating this information to other clients. Note that identifying data in NES environments is a relatively new issue. This is strongly linked to the way we are considering data persistence. In NetSolve, the idea is that data is persistent for a session time and deleted after. In DIET, we think that a data can survive to a session and could be used by other clients than the producer or in later sessions. Nevertheless, a client can also decide that its data are only available in a single session. Currently, as explained before, data identifiers are stored in a file in a client directory.

4.2.3. Logical data manager and physical data manager

In order to avoid interleaving between data messages and computation messages, the proposed architecture separates data management from computation management. The Data Tree Manager is build around two main entities.

4.2.3.1. Logical data manager

The Logical Data Manager is composed of a set of `LocManager` objects. A `LocManager` is set onto the agent with which it communicates locally. It manages a list of couples (data identifier, owner) which represents data that are present in its branch. Hence, the hierarchy of `LocManager` objects provides the global knowledge of the localization of each data.

4.2.3.2. Physical data manager

The Physical Data Manager is composed of a set of `DataManager` objects. The `DataManager` is located onto each SeD with which it communicates locally. It owns a list of `persistent` data. It stores data and has in charge to provide data to the server when needed. It provides features for data movement and it informs its `LocManager` parent of updating operations performed on its data (add, move, delete). Moreover, if a data is duplicated from a server to another one, the copy is set as non persistent and destroyed after it uses with no hierarchy update.

This structure is built in a hierarchical way as shown in Fig. 11. It is mapped on the DIET architecture. There are several advantages to define such a hierarchy. First, communications between agents (MA or LA) and data location objects (LocManager) are local like those between computational servers (SeD) and data storage objects (DataManager). This ensures that this is not costly, in terms of time and network bandwidth, for agents to get information on data location and for servers to retrieve data. Secondly, considering the physical repartition of the architecture nodes (a LA front-end of a local area network for example), when data transfers between servers localized in the same subtree occur, the consequently updates of the infrastructure are limited to this subtree. Hence, the rest of the platform is not involved in the updates.

4.2.4. Data mover

The Data Mover provides mechanisms for data transfers between Data Managers objects as well as between computational servers. The Data Mover has also to initiate updates of DataManager and LocManager when a data transfer has finished.

4.2.5. Client API

A client can specify the persistence mode of its data. This is done when the problem profile is build. Moreover, after the problem has been evaluated by the platform and persistent data are sent or produced, a unique identifier is affected to each data. A client can execute several operations using the identifier:

Table 1
Persistence modes

mode	Description
DIET VOLATILE	not stored
DIET PERSISTENT RETURN	stored on server, movable and copy back to client
DIET PERSISTENT	stored on server and movable
DIET STICKY	stored and non movable
DIET STICKY RETURN	stored, non movable and copy back to client

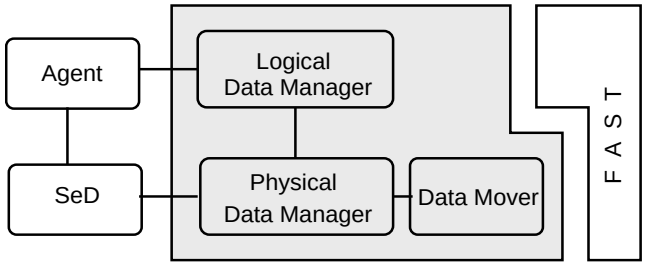


Fig. 10. DTM: data tree manager.

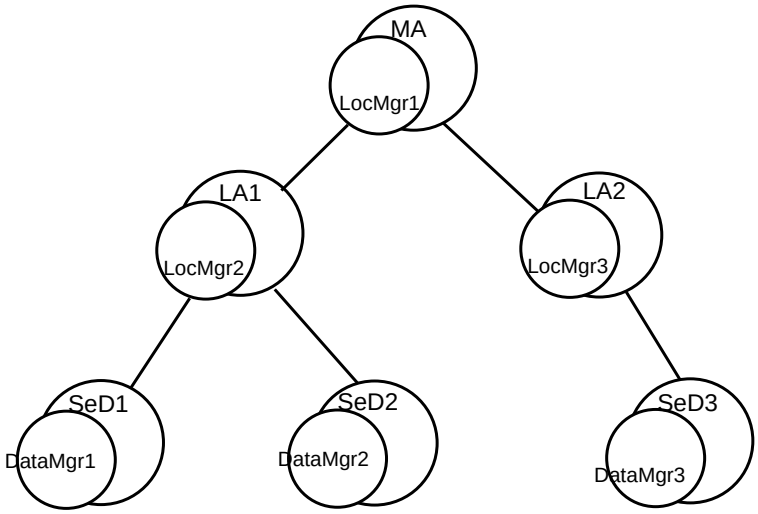


Fig. 11. `DataManager` and `LocManager` objects.

4.2.5.1. Data handle storage

The `store id`() method allows the data identifier to be stored in a local client file. This will be helpful to use data in other session for the same client or for other clients.

```
store_id(char *handle, char *msg);
```

4.2.5.2. Utilization of the data handle

The `diet use data`() method allows the use of a data stored in the platform identified by its handle. The description of the data (its characteristics) is also stored.

```
diet_use_data(char *handle);
```

4.2.5.3. Data remove

The `diet free persistent data`() method allows to free the persistent data identified by `handle` from the platform.

```
diet_free_persistent_data(char
 *handle);
```

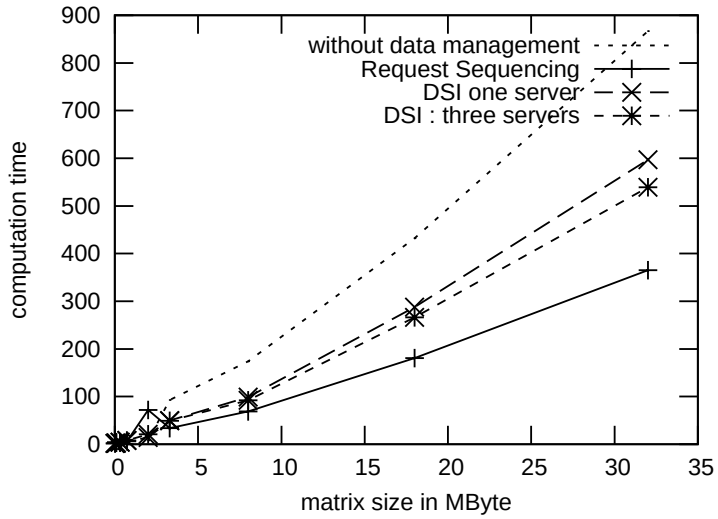


Fig. 12. Standard NetSolve tests.

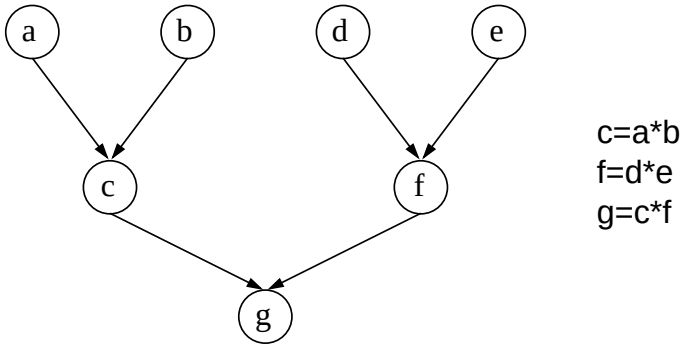


Fig. 13. Matrix multiplication program task graph.

4.2.5.4. Read an already stored data

The `diet read data(char * handle)` method allows to read a data identified by `handle` already stored inside the platform.

```
diet_data_t diet_read_data(char
 *handle);
```

5. Experimental results

5.1. Standard netSolve data management

In this section we test the standard version of NetSolve. We first make experiments on NetSolve without data management and then with the two NetSolve data management approaches described in Section 2.1.2: the Distributed Storage Infrastructure, used to provide data transfer and storage and the Request Sequencing used to decrease network traffic amongst client and servers.

5.2. Experiments

Servers are distributed on a site far from approximatively 100 kilometers to the client. Wide area network is a 16 Mbits/s network while the local area network is an Ethernet 100 Mbits/s network. The platform built for NetSolve tests is composed of three servers, an agent, and an IBP depot.

The experiments consist in a sequence of calls in a session: $C = A * B$ then $D = C + E$ then $A =^t A$. We made three series of test for NetSolve. First, a test using three consecutive blocking calls. Then, a request sequencing test and finally a test with DSI. The last test is divided into two parts: first, a single server

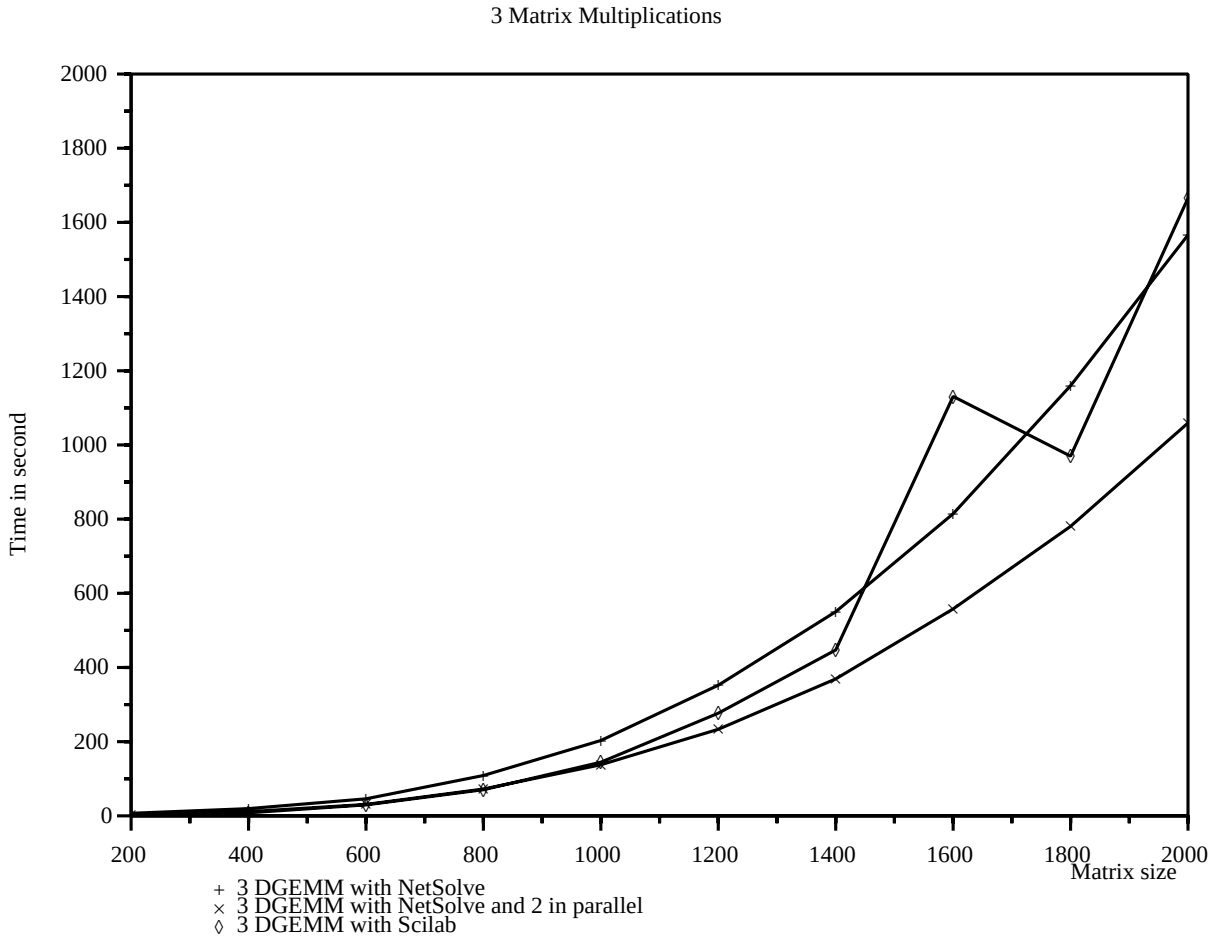


Fig. 14. Matrix multiplications using NetSolve with data persistence on a LAN.

computes all the sequence, then each call is computed by a different server.

Results of the series of tests are exposed in Fig. 12. We note that Request Sequencing is the best solution for such a sequence of calls. When using DSI, we note also that the best solution is when three servers are involved in the computation. This is a bit surprising but it is confirmed by different others tests we made building several different topologies (a server that is also an IBP depot, an IBP depot closest from one server than for the others). In fact, in order to confirm this fact, we try to choose the best server (in terms of processing power and memory capacity) that compute the three calls: but the best solution is always when three servers were involved. We can explain this fact by the memory limitations of the servers involved. A server that have to process three computations does not free its memory implying an overload of this server for further computation.

$$C_{11} = A_{11}B_{11} \; ; \; C_{22} = A_{21}B_{12}$$

$$C_{12} = A_{11}B_{12} \; ; \; C_{21} = A_{21}B_{11}$$

$$C_{11} = C_{11} + A_{12}B_{21} \; ; \; C_{22} = C_{22} + A_{22}B_{22}$$

$$C_{12} = C_{12} + A_{12}B_{22} \; ; \; C_{21} = C_{21} + A_{22}B_{21}$$

Fig. 15. Matrix multiplication using block decomposition.

5.3. NetSolve with data persistence and redistribution

In this section we show several experiments that demonstrate the advantage of using data persistence and redistribution within NetSolve as described in Section 3. Figures 14 and 16 show our experimental results using NetSolve as a NES environment for solving matrix multiplication problems in a grid environment.

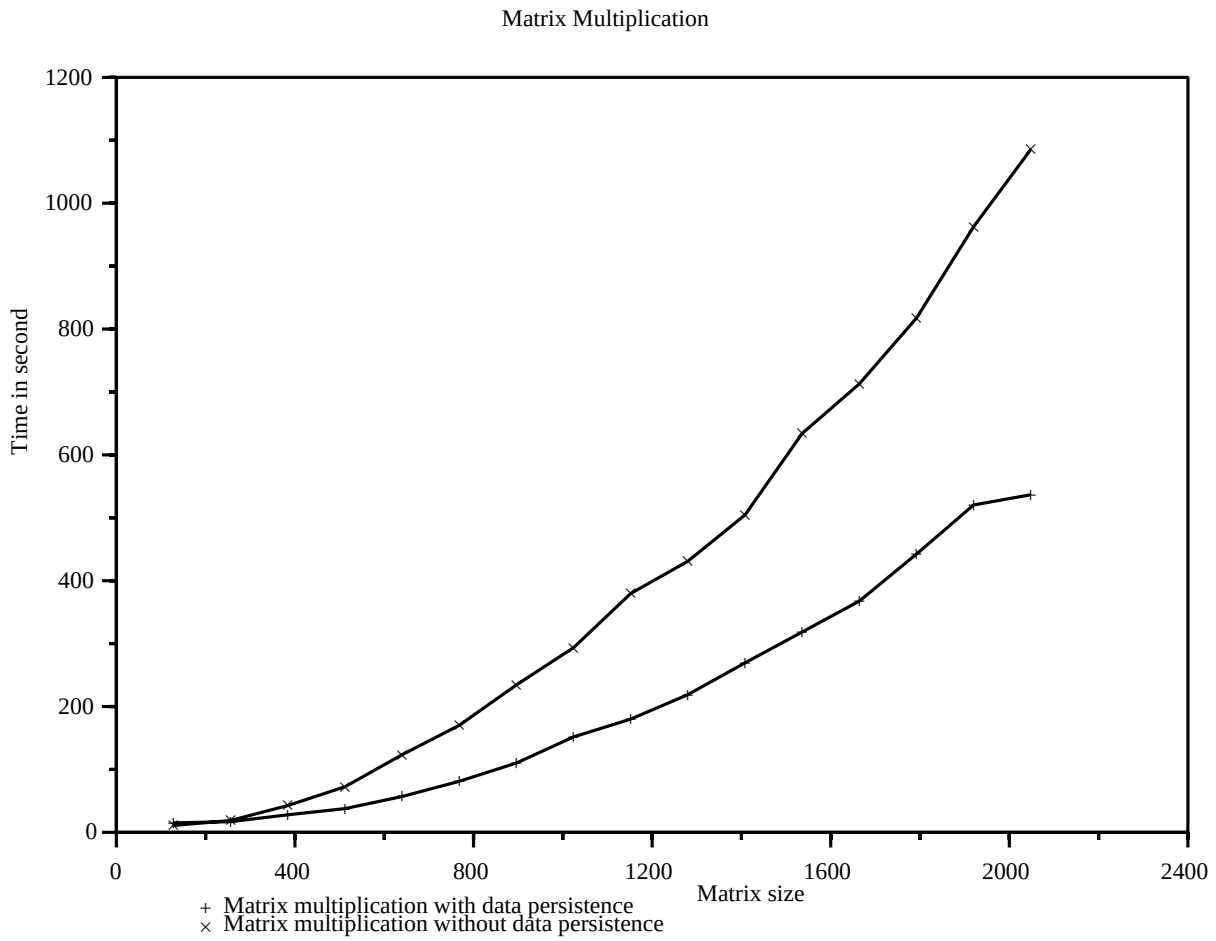


Fig. 16. NetSolve with data persistence: WAN experiments.

5.3.1. LAN experiments

In Fig. 14, 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. Computation and task graphs are shown in Fig. 13. The first two matrix multiplications are independent and can be done in parallel on two different servers. We use Scilab[5] as the baseline for computation time. 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 LAN and that non-blocking calls to servers are helpful for exploiting coarse grain parallelism.

5.3.2. WAN experiments

We have performed a blocked matrix multiplication (Fig. 15). The client and agent were located in one University (Bordeaux) but servers were running on the nodes of a cluster located in Grenoble.[6] The computation decomposition done by the client is shown in Fig. 16. 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 multiplica-

[5] www.scilab.org.

[6] Grenoble and Bordeaux are two French cities separated by about 800 km.

tions 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.

5.4. DIET data management

The first experiments consist in a sequence of calls in a session: $C = A * B$, $D = C + E$ and $A =^t A$. The DIET platform is composed of one MA, two LAs and three servers. Servers are distributed on a site far from approximatively 100 kilometers from the client. The wide area network is a 16 Mbits/s network while the local area network is an Ethernet 100 Mbits/s network. Computers (0.5 Ghz up to 1.8 Ghz) are heterogeneous and run the Linux operating system. We conducted three series of tests: first, a test using three synchronous calls without using DTM. Then, the same sequence using DTM (i.e. using persistence): in this way, A, B, and E matrices are defined as persistent, C matrix must be persistent because it is an input data for the second problem. D matrix can be non persistent because it is not used anywhere else after. Hence, for this case, A, B, E are sent once, and C is not sent. For the last test, only identifiers are sent since all data are already present in the infrastructure.

Results of the series of tests are exposed in Fig. 17. If we can avoid multiple transmissions of the same data, the overall computational time is equal to the transfer time of data into the infrastructure plus the tasks computation time plus the results transfer time to the client. Unsurprisingly again, the last scenario appears to be the best one and confirms the feasibility and the low cost of our approach in the case of a sequence of calls. Using the CORBA space, we can avoid the copy of data by using CORBA memory management methods. These methods allow to get a value without making a memory copy. Moreover, notice that the update of the hierarchy is performed in an asynchronous way, so its cost is very small and does not influence the overall computational time. However, for large data, this approach has the limitations of the memory management.

To complete experiments already lead in [5] and the above results, we have conducted series of tests in order to show the overall advantages of using persistence in DIET. This target architecture is composed of one MA, two LA and two SeD located in a local network. A client is located in a remote site far from 100 kilometers to DIET. The wide area network is a 16 Mbits/s network while the local area network is an Ethernet 100 Mbits/s network. The deployed application is a linear algebra application in which computation time is relatively independent from data size.

In the first experiment, data are in input mode. As seen in Fig. 18, the time of execution varies enormously according to the case. When data is persistent and locally stored onto the computational server, the global execution time is equal to the application computation time. This difference corresponds to the data transfer time profit: approximately 87% for a 400 MBytes matrix. When data is moved between computational servers the gain is of an order of 77% for a 400 MBytes matrix. The difference in gain corresponds to the data transfer time.

In the second experiment, the mode of data is in-out. Profits are less important than for the first experiment, as shown Fig. 19: approximatively 45% for a 400 MBytes matrix if the data is local to the computational server and 40% if the data is moved.

These results confirm the feasibility of our approach and the gains in term of execution time.

5.5. DIET and NetSolve comparison

We summarize here the differences between standard NetSolve, NetSolve with data persistence and redistribution (called NetSolve-PR here), and DIET with data management.

– In standard NetSolve request sequencing approach, the sequence of computations has to be processed by an unique server. In this case, a client needs to have the knowledge of the services provided by a server in order to use this approach. Now, when using DSI, it is useful to have a DSI depot near computational servers in order to decrease transfer time. Hence, the way that DSI architecture is implemented is very important. In NetSolve-PR and in DIET DTM, a client does not need to know which server is able to solve a given problem (considering that a submitted request can

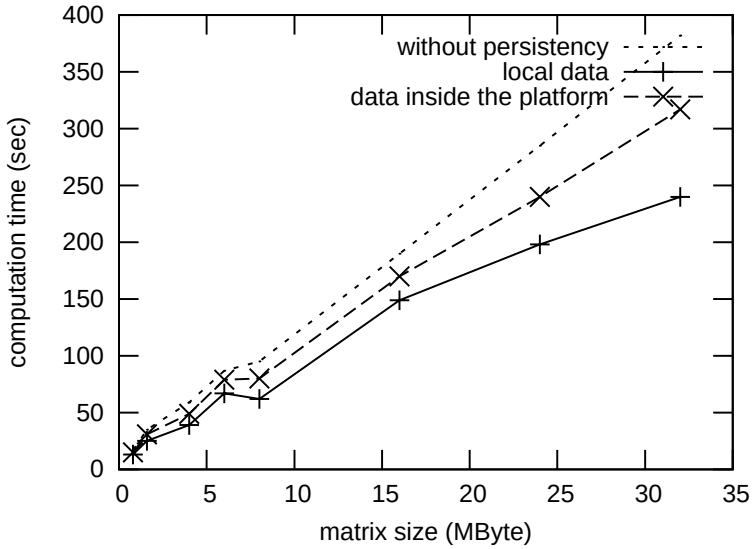


Fig. 17. DIET Tests with and without persistence.

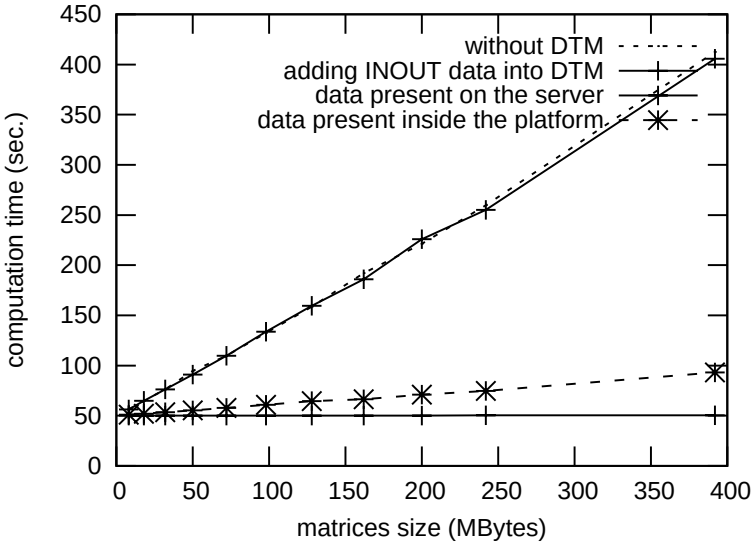


Fig. 18. Sending IN data.

be processed by the platform), and we assume that the data management architecture allows data to be close to the computational server.

– The DIET Data Mover is directly managed by the DTM that allows data to be moved near computational servers. In standard NetSolve with DSI, considering for example two far away computational servers that will need the same data, data must be sent on a DSI depot that is close to each computational server. Hence, data could be sent twice by a client. In NetSolve-PR data always stay on a server and do not use a depot. Data can be sent directly from a client to a server.
– Using NetSolve approach, a client does not need to specify the way its data will be managed. Using request sequencing or DSI, data are considered to be persistent. In DIET DTM, users need to precise the persistence mode of all their data, even for the non persistent ones. NetSolve-PR is backward compatible. This means that when persistence is not needed nothing as to be specified. However, when using persistence the client has to specify it

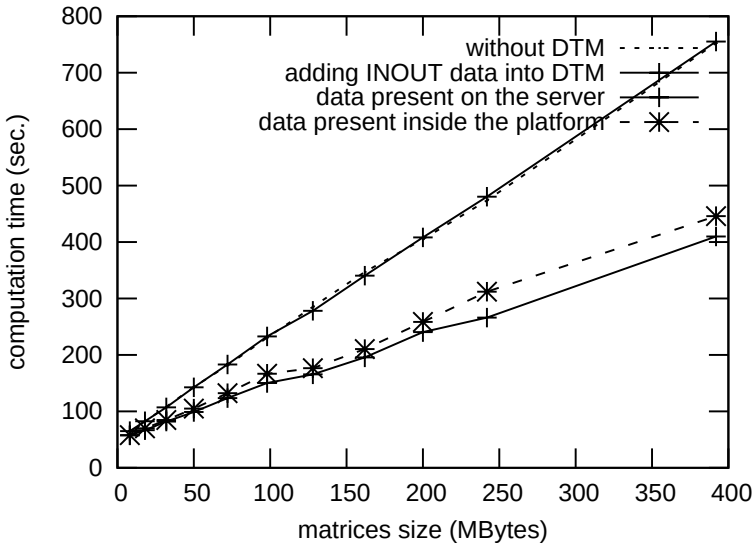


Fig. 19. Sending INOUT data.

in the request.

– In DIET, we think that persistent data must "survive" to a client session and so must be fully identified. Data are kept as long as a client needs it (for later use in other sessions and for other clients in case of collaborative projects for example). In NetSolve (with or without data persistence and redistribution), data are persistent in a session, for a set of computations: data are lost when the client terminate.
– In NetSolve, the system cannot be overloaded by data since data are removed from its depot after computation or removed after a set of computation (request sequencing). In DIET and NetSolve-PR, the way data is managed may lead to a memory overload since data is cached on servers when they are not explicitly send as files.

6. Standardizing data management

As we seen, data management in ASP environments leads to several approaches. However, the need of a common API for ASP environments is essential. Indeed, NetSolve, Ninf and DIET are members of the GridRPC working Group in the GGF which work is to standardize and to implement a remote procedure call mechanism for Grid Computing. This work has already lead to a programming model [20].

Within this GridRPC working group an on-going work supervised by Craig Lee aims at standardizing data management for this model. So far, the proposal is based on two points: a data must be fully identified and a programmer can choose whether a data will be persistent inside the platform or not. This proposal must take into account the different approaches in ASP environments in order to obtain a common layer on which each policy can be integrated.

In order that each data will be fully identified, we define the **data handle** (DH) which is the reference of a data that may reside anywhere. This enables the virtualization of data since it can be read or written without knowing or caring where it is coming from or going to. The creation of a **data handle** is realized by the `create(data handle t *dh);` function.

Once the data reference created, it is also possible to bind it with a data. If data is bound, it must be on the client or on a storage server. Otherwise, data is already stored inside the platform. The bind operation is also used to specify if the data must be keep or not. This operation is realized by the `bind(data handle t dh, data loc t loc, data site t site);` function.

– **data loc t loc** (data location): client side or storage server.
– **data site t site**: location of the machine where data will be stored If (site == NULL) data will be stored on the last computational server (client transparent) If (site == loc) data forwarded to site (client or storage server) If (site <> loc) data moved from loc to site.

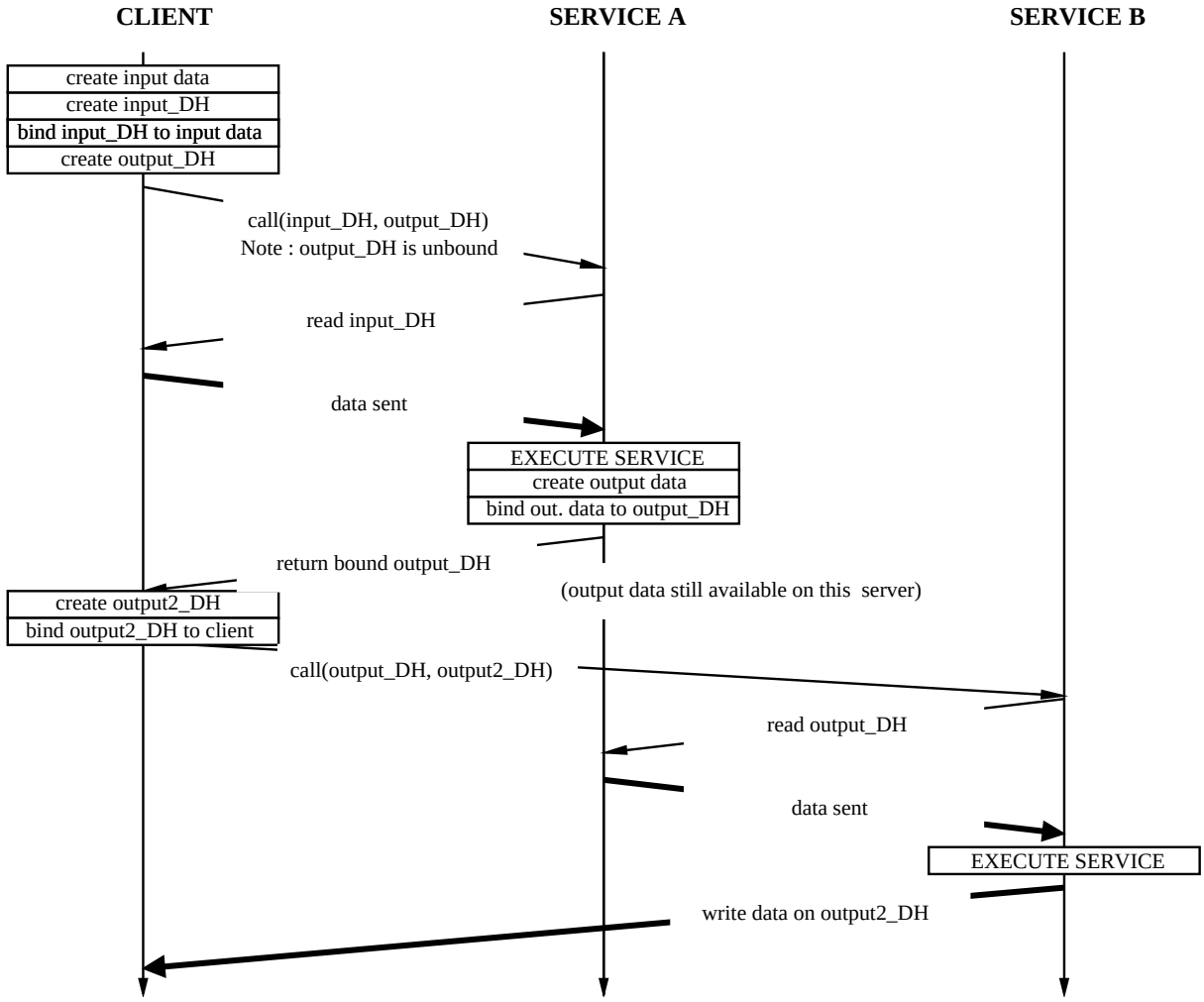


Fig. 20. Using the GridRPC API for data management.

From these to functions, we can define operations on data handles.

- **data t read(data handle t dh)**: read (copy) the data referenced by the DH from whatever machine is maintaining the data. Reading on an unbound DH is an error.
- **write(data t data, data handle t dh)**: write data to the machine, referenced by the DH, that is maintaining storage for it. Writing on an unbound DH could have the default semantics of binding to the local host. This storage does not necessarily have to be pre-allocated nor does the length have to be known in advance.
- **data arg t inspect(data handle t dh)**: Allow the user to determine if the DH is bound, what machine is referenced, the length of the data, and possibly its structure. Could be returned as XML.
- **bool free data(data handle t dh)**: free the data (storage) referenced by the DH.
- **bool free handle(data handle t dh)**: frees the DH.

Figure 20 shows an example of data management within this proposed framework. In this figure, a client submits a problem to a server that is able to compute it and a second problem on an other server. The second server has best performance. For this second computation, the client have not to send data an other time, this data is already in the network.

7. Conclusion and future Work

The litterature proposes several approaches for executing applications on computational grids. The

GridRPC standard implemented in several NES middleware (DIET, NetSolve, Ninf, etc.) is one of most popular paradigm. However, this standard does not define how data can be managed by the system: each time a request is performed on a server, input data are sent from the client to the server and output data are sent back to the client and thus data are not persistent. This implies a large overhead that needs to be avoided. Moreover, no redistribution of persistent data between servers is available. When a data is computed by one server and needed by an other server for the next step of computation it always goes through the client, increasing the transfer time.

In this paper, we have proposed and implemented data management features in two NES (DIET and NetSolve). In NetSolve we changed the internal protocol in order to allow data to stay on server and to move data from one server to an other. We modified the API in order clients to allow data persistence and redistribution and we enhanced the request scheduling algorithm in order to take into account data location. Concerning DIET, we developed a data management service called Data Tree Manager (DTM). This service is based on three key points: a data must be fully identified inside the platform, it must be located and moved between computational servers. The way to think this service was relatively a new concept in NES community. Indeed, our service is able to keep information on data stored as long as the client does not want to remove them.

In our experimental results, we tested our implementations and the standard NetSolve one (which features request sequencing). We shown that data management improves the performance of applications (for both systems) when requests have dependences because it reduces the amount of data that circulates on the Network.

Since we show that the implementation of data management is feasible and it provides an increase of performance, we discuss, in the last section of this article, the standardization proposal (joint work with C. Lee within the GGF) of such a feature. It is based on two points: data is fully and globally identified, and the programmer can choose whether a data is persistent or not in an explicit way.

In our future work, we want to study and propose new scheduling algorithms that efficiently takes into account data management. For instance we believe that a better scheduling algorithm than the proposed enhancement of MCT can be designed in this context.

In the context of DIET, the overview of NetSolve DSI policy leads us thinking about the possibility to keep data on storage servers. The definition of an efficient storage policy will allow to avoid servers overload. Our idea is to keep data onto a server as long as it does not decrease server performance. The data will then be stored in available storage service systems (like IBP).

References

[1] P. Arbenz, W. Gander and J. Moré, The remote computational system, *Parallel Computing* **23**(10) (1997), 1421–1428.

[2] 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, volume volume 1900 of Lecture Notes in Computer Science, pages 1213–1222, Munich Germany, August 2000. Springer Verlag.

[3] E. Caron, S. Chaumette, S. Contassot-Vivier, F. Desprez, E. Fleury, C. Gomez, M. Goursat, E. Jeannot, D. Lazure, F. Lombard, J.M. 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* **27**(11), 2001.

[4] E. Caron and F. Desprez, *DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid*, International Journal of High Performance Computing Applications, 2005. To appear. Also available as INRIA Research Report RR-5601.

[5] E. Caron, F. Desprez, B. Del-Fabbro and A. Vernois, *Gestion de données dans les nes*, In DistRibUtIon de Données à grande Echelle. DRUIDE 2004, Domaine du Port-aux-Rocs, Le Croisic. France, may 2004. IRISA.

[6] E. Caron, F. Desprez, F. Petit and C. Tedeschi, *Resource Localization Using Peer-To-Peer Technology for Network Enabled Servers*, Research report 2004-55, Laboratoire de l'Informatique du Parallélisme (LIP), December 2004.

[7] H. Casanova and J. Dongarra, NetSolve: A Network-Enabled Server for Solving Computational Science Problems, *International Journal of Supercomputer Applications and High Performance Computing* **11**(3) (Fall 1997), 212–213.

[8] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury and S. Tuecke, *The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets*, http://www.globus.org/, 1999. 132.

[9] S. Dahan, J.M. Nicod and L. Philippe, *Scalability in a GRID server discovery mechanism*, In 10th IEEE Int. Workshop on Future Trends of Distributed Computing Systems, FTDCS 2004, pages 46–51, Suzhou, China, May 2004. IEEE Press.

[10] B. Del-Fabbro, D. Laiymani, J. Nicod and L. Philippe, *Data management in grid applications providers*, In Procs of the 1st IEEE Int. Conf. on Distributed Frameworks for Multimedia Applications, DFMA'2005, pages 315–322, Besançcon, France, February 2005.

[11] B. Del-Fabbro, D. Laiymani, J.-M. Nicod and L. Philippe, A data persistency approach for the diet metacomputing environment, in: *International Conference on Internet Computing*, H.R. Arabnia, O. Droegehorn and S. Chatterjee, eds, Las Vegas, USA, June 2004. CSREA Press, pp. 701–707.

[12] GridRPC Working Group, https://forge.gridforum.org/projects/gridrpc-wg/.

[13] S. Sekiguchi, H. Nakada and M. Sato, Design and Implementations of Ninf: Towards a Global Computing Infrastructure. Future Generation Computing Systems, *Metacomputing Issue* **15** (1999), 649–658.

[14] T. Kosar and M. Livny, *Stork: Making data placement a first class citizen in the grid*, 2004. In Proceedings of the 24th Int. Conference on Distributed Computing Systems, Tokyo, Japan, March 2004.

[15] 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.

[16] S. Matsuoka, H. Nakada, M. Sato and S. Sekiguchi, *Design Issues of Network Enabled Server Systems for the Grid*, http://www.eece.unm.edu/~dbader/grid/WhitePapers/satoshi.pdf, 2000. Grid Forum, Advanced Programming Models Working Group whitepaper.

[17] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee and H. Casanova, *A GridRPC Model and API for End-User Applications*, December 2003. https://forge.gridforum. org/projects/gridrpc-wg/document/GridRPC EndUse%r16dec03/en/1.

[18] NEOS. http://www-neos.mcs.anl.gov/.

[19] M. Quinson, *Dynamic performance forecasting for network-enabled servers in a meta- computing environment*, In International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS'02), in conjunction with IPDPS'02, April 15–19 2002.

[20] K. Seymour, C. Lee, F. Desprez, H. Nakada and Y. Tanaka, *The End-User and Middleware APIs for GridRPC*, In Workshop on Grid Application Programming Interfaces, In conjunction with GGF12, Brussels, Belgium, September 2004.