In: Book Title
Editor: Editor Name, pp. 141-181
ISBN 0000000000

Chapter 1

HIGH PERFORMANCE GRIDRPC MIDDLEWARE

Yves Caniou, [*] ***Eddy Caron,*** [†] ***and Frédéric Desprez*** [‡]
Université de Lyon, LIP, CNRS-ENS-Lyon-UCBL-INRIA, France
Hidemoto Nakada [§] ***and Yoshio Tanaka*** [¶]
National Institute of Advanced Science and Technology,
1-1-1 Umezono, Tsukuba, Ibaraki 305-8568, Japan
Keith Seymour [‖]
Electrical Engineering and Computer Science Department,
University of Tennessee, Knoxville, TN, USA

Abstract

A simple way to offer a Grid access through a middleware is to use the GridRPC paradigm. It is based on the classical RPC model and extended to Grid environments. Client can access to remote servers as simply as a function call. Several middlewares are compliant to this paradigm as DIET, GridSolve, or Ninf-G. Actors of these projects have worked together to design a standard API within the Open Grid Forum. In this chapter we give an overview of this standard and the current works around the data management. Three use cases are introduced through a detailled descriptions of DIET, GridSolve, and Ninf-G middleware features. Finally applications for each middleware are shown to appreciate how they take benefit of the GridRPC API.

Key Words: GridRPC, Network Enabled Systems, Programming API

[*] UCBL, E-mail address: Yves.Caniou@ens-lyon.fr
[†] ENS Lyon, E-mail address: Eddy.Caron@ens-lyon.fr
[‡] INRIA, E-mail address: Frederic.Desprez@inria.fr
[§] E-mail address: hide-nakada@aist.go.jp
[¶] E-mail address: yoshio.tanaka@aist.go.jp
[‖] E-mail address: seymour@cs.utk.edu

1 Introduction

Large problems coming from numerical simulation or life science can now be solved through the Internet using grid middleware [10, 27]. Transparency and ease of use is sometimes more important for a user than raw performance. Along with researches and development around middleware and services for grids, researchers and developers have been working on programming issues of large scale distributed systems. Several approaches co-exist to port application on grid platforms like classical message-passing [33, 38], batch processing [57, 67], web portals [28, 30, 32], workflow management systems [23, 29, 45, 70], object-oriented approaches [7, 68].

Among existing middleware and application programming approaches [34], one simple, powerful, and flexible approach consists in using servers available in different administrative domains through the classical client-server or Remote Procedure Call (RPC) paradigm. Network Enabled Servers (NES) [35] implement this model, which is also called GridRPC [53]. Clients submit computation requests to a scheduler whose goal is to find a server available on the grid. Scheduling is frequently applied to balance the work among the servers and a list of available servers is sent back to the client; the client is then able to send the data and the request to one of the suggested servers to solve its problem. Thanks to the growth of network bandwidth and the reduction of network latency, small computation requests can now be sent to servers available on the grid. To make effective use of today's scalable resource platforms, it is important to ensure scalability in the middleware layers as well. This service oriented approach is not new.

Several research projects have targeted this paradigm in the past. The main middlewares implementing the API are the ones presented here, i.e. NetSolve/GridSolve, Ninf, and DIET but some other environments support it like OmmiRPC [50], XtremWeb [14], and the SAGA interface from the OGF. The RPC model over the internet has also been used for several applications. In [24], the authors describe the use of remote computations available as services for optmization problems. Transparently through the Internet, large optimization problems can be solved using different approaches by simply filling a web page. Remote mage processing computations are described in [5] and mathematical libraries in [12]. Some systems target clusters like OVM [11] or they can be linked with languages like OpenMP [51]. Even P2P systems can be used using this model like with XtremWeb [14] and also having fault-tolerance embedded in the middleware itself [21]. This approach of providing computation services through the Internet is also highly close to the Service Oriented Computing (SOA) paradigm [44].

The goal of this chapter is first to describe the API itself and its extensions for data management in Section 2. Then, in Sections 3, 4 and 5, we review three existing implementations over heavily used middleware platforms. This presentation allows to understand how performance can be obtained from these middleware using the GridRPC API. Finally, before a conclusion, we present in Section 6 applications from different fields ported using the API over several grids.

2 GridRPC API Presentation

One simple, yet effective, mean to execute jobs on a computing grid is to use a GridRPC middleware, which relies on the GridRPC paradigm. Numerous implementations are currently available, such as DIET [20], NetSolve [69], Ninf [60], OmniRPC [50].

For each request, the GridRPC middleware manages the management of the submission, of the input and output data, of the execution of the job on the remote resource, etc. To make available a service, a programmer must implement two codes: a client, where data are defined and which is run by the user when requesting the service, and a server, which contains the implementation of the service which is executed on the remote resource.

One step to ease the development of such codes conducted to define a GridRPC API [39], which has been proposed as a draft in September 2004 and which is an Open Grid Forum (OGF) standard since September 2007. Thus a GridRPC source code can be compiled and executed with any GridRPC compliant middleware.

Due to the difference in the choice of implementation of the GridRPC API, a document describing the interoperability between GridRPC middleware has been written [62]. Its main goals are to describe the difference in behaviour of the GridRPC middleware and to propose a common test that all GridRPC middleware must pass. Nevertheless, it is not of its purpose to make a common interoperable client, which could participate to different grid middleware at the same time. This document is intended to be soon an Open Grid Forum standard.

Discussions are currently undertaken on the data management within GridRPC middleware. A draft of an API has been proposed during the OGF'21 in October 2007. The motivation for this document is to provide explicit functions to manipulate the data exchange between a GridRPC platform and a client since (1) the size of the data used in grid applications may be large and useless data transfers must be avoided; (2) data are not always stored on the client side but may be made available either on a storage resource or within the GridRPC platform.

In the following, we first describe the GridRPC API and the efforts concerning the interoperability between implementations, and then we present the current work on defining an API for GridRPC data management.

2.1 The GridRPC API and Interoperability Between Implementations

One of the goals of the GridRPC API is to clearly define the syntax and semantics for GridRPC, which is the extension of the Remote Procedure Call (RPC) to grid environments. Hence, end-user client/server applications can be written given the programming model.

2.1.1 The GridRPC Paradigm

The GridRPC model is pictured in Figure 1: (1) servers register their services to a registry; (2) when a client needs the execution of a service, it contacts the registry and (3) the registry returns a handle to the client; (4) then the client uses the handle to invoke the service on the server and (5) eventually receives back the results.

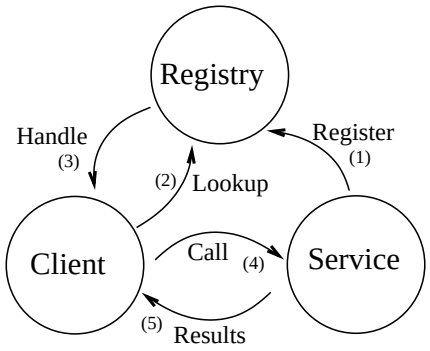


Figure 1: The GridRPC model.

2.1.2 The GridRPC API

Mechanisms involved in the API must provide means to make synchronous and/or asynchronous calls to a service. If the latter, clients must also be able to wait in a blocking or non-blocking manner after the completion of a given service. This naturally involves some data structures and conducts to a rigorous definition of the functions of the API.

GridRPC Data Types

Three main data types are needed to implement the API: (1) **grpc_function_handle_t** is the type of variables representing a remote function bound to a given server. Once allocated by the client, such a variable can be used to launch the service as many times as desired. It is explicitly invalidated by the user when not needed anymore; (2) **grpc_session_t** is the type of variables used to identify a specific non-blocking GridRPC call. Such a variable is mandatory to obtain information on the status of a job, in order for a client to wait after, cancel or know the error status of a call; (3) **grpc_error_t** groups all kind of errors and returns status codes involved in the GridRPC API.

GridRPC Functions

grpc_initialize() and **grpc_finalize()** functions are similar to the MPI initialize and finalize calls. It is mandatory that any GridRPC call is performed in between these two calls. They read configuration files, make the GridRPC environment ready and finish it.

In order to initialize and destruct a function handle, **grpc_function_handle_init()** and **grpc_function_handle_destruct()** functions have to be called. Because a function handle can be dynamically associated to a server, because of resource discovery mechanisms for example, a call to **grpc_function_handle_default()** let to postpone the server selection until the actual call is made on the handle.

grpc_get_handle() let the client retrieve the function handle corresponding to a session ID (*e.g.,* to a non-blocking call) that has been previously performed.

Depending on the type of the call, blocking or non-blocking, the client can use the **grpc_call()** and **grpc_call_async()** function. If the latter, the client possesses after the call a session ID which can be used to respectively probe or wait for completion, cancel the call

and check the error status of a non-blocking call.

After issuing a unique or numerous non-blocking calls, a client can use: **grpc_probe()** to know if the execution of the service has completed; **grpc_probe_or()** to know if one of the previous non-blocking calls has completed; **grpc_cancel()** to cancel a call; **grpc_wait()** to block until the completion of the requested service; **grpc_wait_and()** to block until all services corresponding to session IDs used as parameters are finished; **grpc_wait_or()** to block until any of the service corresponding to session IDs used as parameters has finished; **grpc_wait_all()** to block until all non-blocking calls have completed; and **grpc_wait_any()** to wait until any previously issued non-blocking request has completed.

2.1.3 Presentation of the Interoperability Between Implementations

The Open Grid Forum standard describing the GridRPC API did not focus on the implementation of the API. Then, divergences in implementation have been observed. In order to make a GridRPC client-server code reusable in all GridRPC middleware relying on the GridRPC API, a work has been tackled to propose interoperability between implementations.

The paper [62] points out, with exhaustive test-cases, the differences and convergences in behavior of the main GridRPC middleware implementations, namely DIET, Netsolve, and Ninf. In addition, a program has been written to test the GridRPC compliance of all middleware.

2.2 GridRPC Data Management API

The data management extension is designed to provide a way to explicitly manage the data and their placement in the GridRPC model. With the help of this explicit data management, the client will avoid useless transfers of large data. However, the client may not want to, or may not know how to manage data. Then, the default behavior of the GridRPC Data Management extension must be standardized.

In a GridRPC environment, data can be stored either on a client host, on a data storage server, on a computational server or inside the GridRPC platform. When clients do not need to manage their data, then the basic GridRPC API is sufficient. On each `grpc_call()`, data is transferred between a client and the computational server used. Once the computation performed, results are sent back to the client. However, to minimize data transfers, clients need data management functions. We can consider two kinds of data: (a) external data and (b) internal data. External data are placed on servers, like data repositories. These servers are not registered inside the platform but can be directly accessed to read/write data. The use of such data implies several data transfers if the client uses the basic GridRPC API: the client must download the data and then send it to the GridRPC platform when issuing the call to `grpc_call()`. One of these transfers should be avoided: the client may just give a data reference (also called *handle*) to the platform/server and the transfer is completed by the platform/server. Examples of such Data Storage servers are IBP [46] and SRB [6]. Among the different available examples of this approach in GridRPC environment, we can cite the Distributed Storage Infrastructure of NetSolve [8] or the utilization of JuxMem in DIET [3]. Internal data are managed inside the GridRPC platform. Their

placement depends on computations and it may be transparent to clients: in this case, the GridRPC middleware can manage data. Temporary data, generated by request sequencing [4], are examples of internal data. For instance, a client issues two calls to solve the same problem and the second call uses input or output data from the first call. Other cases of useless temporary data occur when the results of a simulation are sent to a graphical viewer as done in most Problem Solving Environments (PSE). Among the examples of internal data management, we can cite the Data Tree Management infrastructure (DTM) used in DIET [19], and the data layer OmniStorage in OmniRPC [1]. This approach is suitable for, but not limited to, intermediate results to be reused in case of request sequencing.

In both cases, it is mandatory to identify each data. All data stored either in the platform or on storage servers will be identified by **Data Handles** and **Storage Information**. Without lack of generality, we define the *GridRPC data type* as either *the data used for a computational problem*, either both a *Data Handle* and *storage information*. Indeed, when a computational server receives a GridRPC data which does not contain the computational data, it must know the unique name of the data with the Data Handle, and must know its location to get it and where the client wants to save it after the computation. Thus storage information must record the original location of the data and the destination of the data.

In [39], data used as input/output parameters are provided within the `<varargs>` notation of the `grpc_call()` and `grpc_call_async()` functions. Without lack of generality, and in order to propose an API independent of the language of implementation, we refer to `grpc_data_t` as the type of such variables. Thus, in the following, a `grpc_data_t` is any kind of data, or contains a reference on the computational data, which we call a *Data Handle*, as well as some *Storage Information*.

The GridRPC data includes at least the data or a data handle, and may contain some information about the data itself (*e.g.*, type, size) as well as information on its location and the protocol used to access it (*e.g.*, the URI of a specific server, a link with a Storage Resource Broker, containing the correct protocol to use). A data handle is essentially a unique reference to a data that may reside anywhere. Data and data handles can be created separately. By managing GridRPC data with data handles, clients do not have to know where data are currently stored.

2.2.1 GridRPC Data Management Data Type

A data in a GridRPC middleware is defined by the **grpc_data_t** type. It relies on a data, or on a grpc_data_handle_t type and a grpc_data_storage_info_t type to access it. Consequently, the grpc_data_t type can be seen as a structure containing the data itself and/or a grpc_data_handle_t. The grpc_data_storage_info_t type can also be stored in the grpc_data_t structure or it can also be stored and managed inside the GridRPC data middleware.

A variable of the **grpc_data_handle_t** type represents a specific data. It is allocated by the user. After a *data handle* was initialized, it may be used in a server invocation. The lifetime of a *data handle* is determined when the user invalidates it. Data handles are created/allocated by simply creating a variable of this type.

Variables with **grpc_data_storage_info_t** type represent information on a specific data which can be local or remote. It is at least composed of: (1) Two URIs, one to access the

data and one if the data has to be stored somewhere from this server (for example, an OUT parameter to transfer at the end of a computation); (2) Information concerning the mode of management. For example, data management is defaulted to the one of the standard GridRPC paradigm, but it can be noted for example as GRPC_PERSISTENT, which corresponds to a transparent management by the GridRPC middleware, or GRPC_STICKY, in which case the data cannot migrate but can be replicated; (3) Information concerning the type of the data, as well as its size.

2.2.2 GridRPC Data Management Functions

The **grpc_data_init()** function initializes the *GridRPC data* with a specific data. This data may be available locally or on a remote storage server. Both identifications can be used. GridRPC data referencing input parameters must be initialized with identified data before being used in a `grpc_call()`. GridRPC data referencing output parameters do not have to be initialized. The function **grpc_data_getinfo()** let the user access information about the grpc_data_t. It returns information on data characteristics, status, and location.

The **grpc_data_write()** function writes a GridRPC data to the output location set during the init call in the output parameters fields. For commodity reasons, a diffusion mode and a list of additional servers on which the data has to be uploaded can be provided. In that case, the protocol defined during the init call is used. Some broadcast/multicast mechanisms can then be implemented in the GridRPC data middleware in order to improve performance. The diffusion mode can be used by more intelligent data middleware to diffuse a data in a broadcast manner for example. A dual function called **grpc_data_read()** is available. After calling this function, the data will be available in the GridRPC data type `grpc_data_t`, which will also still contain the data handle.

The **grpc_unbind_data()** function is used by a client when it does not need the handle on the GridRPC data anymore. To explicitly erase the data on a storage resource, the client can call the **grpc_free_data()** function which frees the GridRPC data.

In order to communicate a reference between grid users, for example in case of large size data, one should be able to store a GridRPC data. The location can then be shared, for example by mail, an consequently the GridRPC data management API proposes two functions: **grpc_data_load()** and **grpc_data_save()**.

2.3 GridRPC Example

In the example described in Figure 2, we show how to re-use data on a specific server without resending them. Client wants to compute $C = C \times A^n$ using the service `"*"` on server *karadoc*. It shows how to use the GridRPC data management functions when the data needs to be stored inside the platform, to keep the data on the same server, with the help of the GRPC_STICKY mode.

2.3.1 Input Data

Data `A` will be used and will remain on server *karadoc*. We can use the GRPC_STICKY parameter to keep the data on server *karadoc*. Data `C` is an input/output data. The first

```
grpc_function_handle_init(handle13,"karadoc.aist.go.jp","*");
grpc_data_init(&dhA,"LOCAL_MEMORY://britannia.ens-lyon.fr/&A","LOCAL_MEMORY://karadoc.aist.go.jp",
               GRPC_DOUBLE, GRPC_STICKY);
grpc_data_init(&dhC, "NFS://britannia.ens-lyon.fr/home/user/C.in", "LOCAL_MEMORY://karadoc.aist.go.jp",
               GRPC_DOUBLE, GRPC_STICKY);

for(i=0;i<n+1;i++)
{
   if( i==1 )
      grpc_data_init(&dhC, "LOCAL_MEMORY://karadoc.aist.go.jp",NULL, DOUBLE, STICKY);
   if( i==n )
      grpc_data_init(&dhC, "LOCAL_MEMORY://karadoc.aist.go.jp","NFS://britannia.ens-lyon.fr/home/user/C.out",
                     GRPC_DOUBLE, GRPC_VOLATILE);

   grpc_call(handle1,dhA,dhC,dhC);
}
grpc_data_free(dhA);
grpc_data_free(dhC);
```

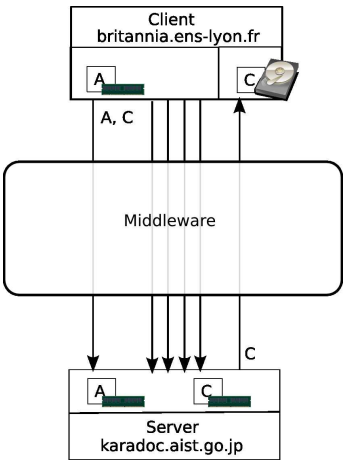


Figure 2: GridRPC call with data management using persistence through the GRPC_STICKY mode.

`grpc_data_init` for this data requires only an input location and the GRPC_STICKY mode.

2.3.2 Output Data

Output data `C` is generated on server `karadoc` but only the last result is useful for the client. Thus, to send the final result to the client we update the output location just before the last `grpc_call()`.

3 DIET

The Distributed Interactive Engineering Toolbox (DIET) [15, 20] project is focused on the development of a scalable middleware with initial efforts focused on the distribution of the scheduling problem across multiple agents. DIET consists of a set of elements that can be used together to build applications using the GridRPC paradigm. This middleware is able to find an appropriate server according to information given in the client request (*e.g.*, problem to be solved, size of the data involved), the performance of the target platform (*e.g.*, server load, available memory, communication performance) and the local availability of data stored during previous computations. The scheduler is distributed using several

collaborating hierarchies connected either statically or dynamically (in a peer-to-peer fashion). Data management is provided to allow persistent data to stay within the system for future re-use. This feature avoids unnecessary communication when dependencies exist between different requests (*e.g.*, in case of same or different requests using same data will be executed on the same server). Servers have the possibility to launch several tasks in a time-shared manner, or sequentially, making servers buffer some work [18] or on batch systems.

3.1 DIET Architecture

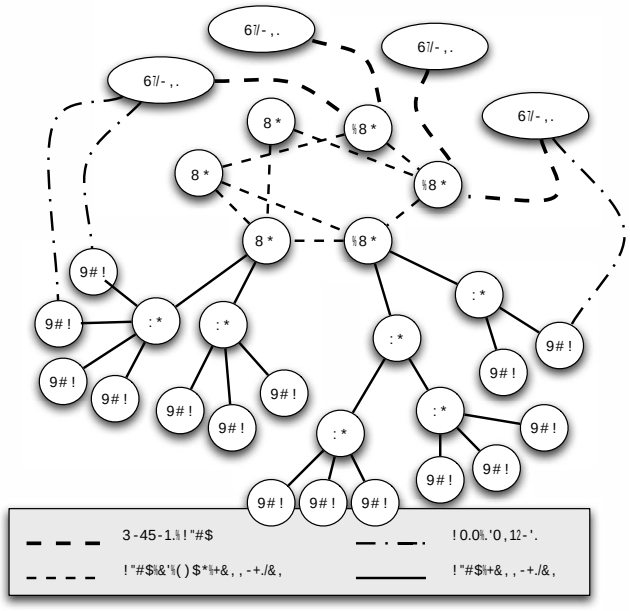


Figure 3: DIET hierarchical organization.

The DIET architecture is hierarchical for a better scalability. The architecture provides flexibility and can be adapted to diverse environments including heterogeneous network hierarchies. DIET is implemented in CORBA and thus benefits from the many standardized, stable services provided by freely-available and high performance CORBA implementations. DIET is based on several components. A **Client** is an application that uses DIET to solve problems using an RPC approach. Users can access DIET via different kinds of client interfaces: web portals, PSEs such as Scilab, or from programs written in C or C++. A **SED**, or server daemon, provides the interface to computational servers and can offer any number of application specific computational services. A SED can serve as the interface and execution mechanism for a stand-alone interactive machine, or it can serve as the interface to a parallel supercomputer by providing submission services to a batch scheduler.

Agents provide higher-level services such as scheduling and data management. These services are made scalable by distributing them across a hierarchy of agents composed of a single **Master Agent (MA)** and any number of **Local Agents (LAs)**. Each DIET hierarchy is independent but the MA can connect to other MAs either statically or in a peer-to-peer

fashion to access resources available via other other hierarchies. Figure 3 shows an example of several DIET hierarchies.

A **Master Agent** is an entry point of our environment. In order to access DIET scheduling services, clients only need a string-based name for the MA (*e.g.*, "MA1") they wish to access; this MA name is matched with a CORBA identifier object via a standard CORBA naming service. Clients submit requests for a specific computational service to the MA. The MA then forwards the request in the DIET hierarchy and the child agents, if any exist, forward the request onwards until the request reaches the SEDs. SEDs then evaluate their own capacity to perform the requested service; capacity can be measured in a variety of ways including an application-specific performance prediction, general server load, or local availability of data-sets specifically needed by the application. SEDs forward their responses back up in the agent hierarchy. Agents perform a distributed collation and reduction of server responses until finally the MA returns to the client a list of possible server choices sorted using an objective function such as computation cost, communication cost, or machine load. The client program may then submit the request directly to any of the proposed servers, though typically the first server will be preferred as it is predicted to be the most appropriate server. The client can submit several simultaneous requests through the use of threading computation in the client code. However, the synchronous mode is not the only request mode. The client can also use the asynchronous mode to submit requests to the DIET hierarchy. When submitting an asynchronous request, the client will not wait the end of the call. To be sure that the request has been well computed the user can use "barriers" to wait for one or all of the ended submitted requests. The scheduling strategies used in DIET are described in Section 3.2.

3.2 DIET SCHEDULING

3.2.1 Plug-in Schedulers

DIET provides a special feature for scheduling requests through its plug-in schedulers. As the applications that are to be deployed on the grid vary greatly in terms of performance demands, the DIET user is provided with the possibility of defining requirements for the scheduling of tasks by configuring the appropriate scheduler.

Application developers may also define performance values to be included in a SED response to a client request. For example, a DIET SED that provides a service to query particular databases may need to include information about which databases are currently resident in its disk cache so that data transfer times can be minimized. Application developers can define their own performance estimation routine or function when developing the application-specific portion of the SED. At this point, any services added to the SED will be associated with the performance estimation routine declared.

Application developers can define their own performance estimation routine or function when developing the application-specific portion of the SED. At this point, any services added to the SED will be associated with the performance estimation routine declared.

For scheduling step, DIET needs reliable resource information from grid resource information services. The performance estimation values required for plug-in schedulers are stored in a performance estimation vector. Information are provided by the SEDs as a response to a client call propagated from the master agent to local agents and finally to the

server level. The SEDs use CoRI (**Collector of Resource Information**) to fill this vector.

CoRI is designed to add any new monitoring tool interface or even any new prediction tool within DIET. It could be dangerous to rely on a single prediction tool for all resource information needs. For example, the prediction tool may not be available on a given architecture and the software dependencies may fail or be too difficult to satisfy in a particular environment. In this case, the scheduler does not receive enough information. This tool must always provide an answer in order to avoid the failure of the whole grid system. If the tool is not able to provide a measurement, a generic response must be provided. Finally, the tool must provide one single interface for all kinds of resource information services. If the environment does not provide a prediction tool we propose a feature which provides a basic set of performance measurements that can satisfy basic scheduler needs. Then the service developer can rely on this collector of resource information (called CoRI-Easy) even if no other resource services like NWS, Ganglia, etc., are available. Moreover, the tool must manage the use of different collectors at the same time and in a similar way. The **CoRI Manager** was designed for this management for the second problem, namely management of different collectors.

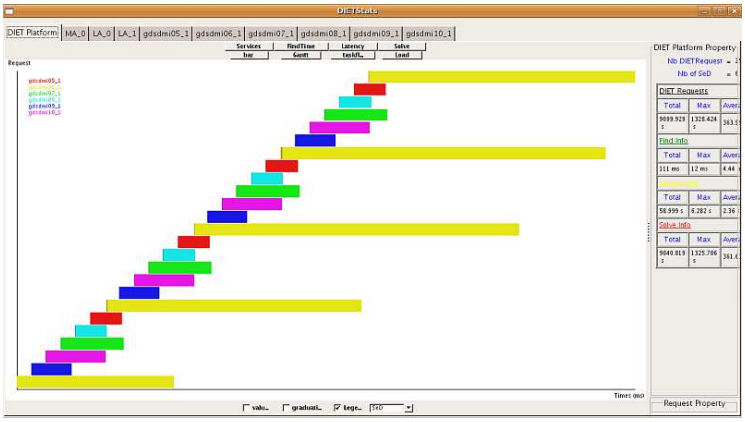
To conclude this section, Figure 4 shows an experiment using two types of scheduler. The first scheduler uses a simple round robin algorithm wherein we have six servers and round robin works on a rotating basis so that one server is assigned some work, then moves to the back of the list. The second scheduler is a CPU scheduler that maximizes the ratio of $\frac{BOGOMIPS}{1+load\_average}$. This experiment is intended to be a proof of the utility of CoRI and the plug-in schedulers with respect to the round robin scheduling scheme existing before their development, as well as a proof of concept in general for the facility of tunable scheduling schemes offered by DIET.

The behavior of both schedulers was studied for requests with different inter-arrival times on a heterogeneous cluster. In this paper we focus on 1 minute for the request inter-arrival time in order to see how the CPU scheduler performs when sufficient time is provided for an accurate estimation of the load average. The distribution of the tasks for the CPU scheduler was performed only on the four fastest nodes resulting in quasi-equal small times for all the tasks. In the case of the Round Robin scheduler, some tasks were privileged by being assigned to the fastest servers while others required longer computing times because all servers were used and some were slower. The total computation time on the platform is smaller with the CPU scheduler due to the fact that faster servers are more utilized. The overlap of tasks observed in the case of the Round Robin scheduler on the slowest processor resulted in larger computing times.

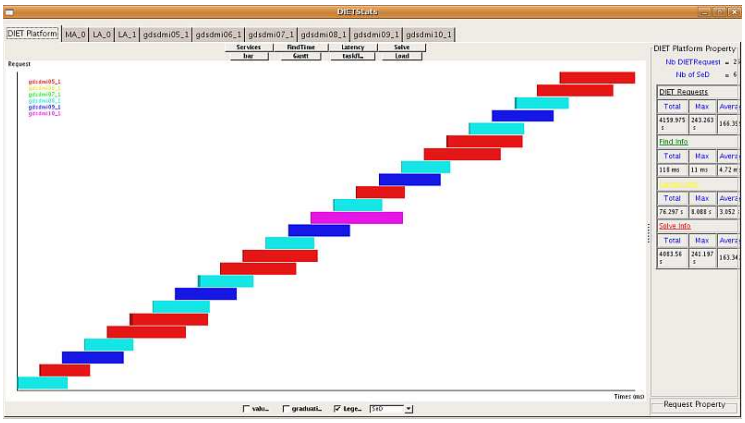
3.2.2 DIET Batch Scheduler Management

Parallel grid resources (parallel machines or clusters of workstations) are generally managed by a reservation batch system such as Loadleveler[1], PBS[2], or OAR[3]. Such a system is responsible for managing the submitted jobs and locating and allocating the required resources. It accepts user submission scripts which must normally contain a variety of in-

[1] http://www-03.ibm.com/servers/eserver/clusters/software/loadleveler.html
[2] http://www.clusterresources.com/pages/products/torque-resource-manager.php
[3] http://oar.imag.fr/



(a) Round Robin Scheduler



(b) CPU Scheduler

Figure 4: Comparison between the taskflows for 25 consecutive requests with task inter-arrival time equal to 1 minute.

formation including the requested number of resources and the amount of time needed for the reservation (walltime).

An efficient grid middleware should provide *transparent access* to parallel resources for the user. It must choose the best parallel resource that suits the request, eventually provide for the parallel malleable task the right number of processors, provide the corresponding walltime, and submit this information to the batch system in an automatically built script in the language of the reservation system. Indeed, as the user does not need to know where his/her job is executed (so the computation availability, etc.), such a script should be produced by the middleware in place of the user.

DIET has the possibility to submit jobs to batch systems including Loadlever and OAR systems. The DIET parallel/batch API provides several functions on both client and server side. On the client side, the client can explicitly ask for a sequential/parallel computation of its job, but otherwise and whenever possible, DIET will choose the best available allocation among sequential/parallel resources. On the server side, the SED programmer builds a script that is generic for all batch schedulers: the DIET server API provides generic envi-

ronment variables to perform the necessary abstraction to the site where the job is executed.

3.2.3 DIET Workflow Management

A large number of scientific applications are represented by graphs of tasks which are connected based on their control and data dependencies. The workflow paradigm on grids is well adapted for representing such applications and the development of several workflow engines [2, 43, 54, 63] illustrate significant and growing interest in workflow management within the grid community. The success of this paradigm in complex scientific applications can be explained by the ability to describe such applications in high levels of abstraction and in a way that makes it easy to understand, change, and execute them.

Several techniques have been established in the grid community for defining workflows. The most commonly used model is the graph and especially the directed acyclic graph (DAG). Since there is no standard language to describe scientific workflows, the description language is environment dependent and usually XML based, though some environments use scripts. In order to support workflow applications in the DIET environment, we have developed and integrated a workflow engine. Our approach has a simple and a high level API, the ability to use different advanced scheduling algorithms, and it should allow the management of multi-workflows sent concurrently to the DIET platform.

DIET users, following the GridRPC paradigm, usually submit individual tasks. Workflows can of course be decomposed in individual tasks but the knowledge of the overall structure of the graphs helps the scheduler to make wise mapping decisions. Thus we extended the agent hierarchy by adding a new special agent to handle workflow submissions. This special agent, called a MA_{DAG}, manages the different workflow submissions. An overview of the extended DIET architecture is shown in Figure 5.

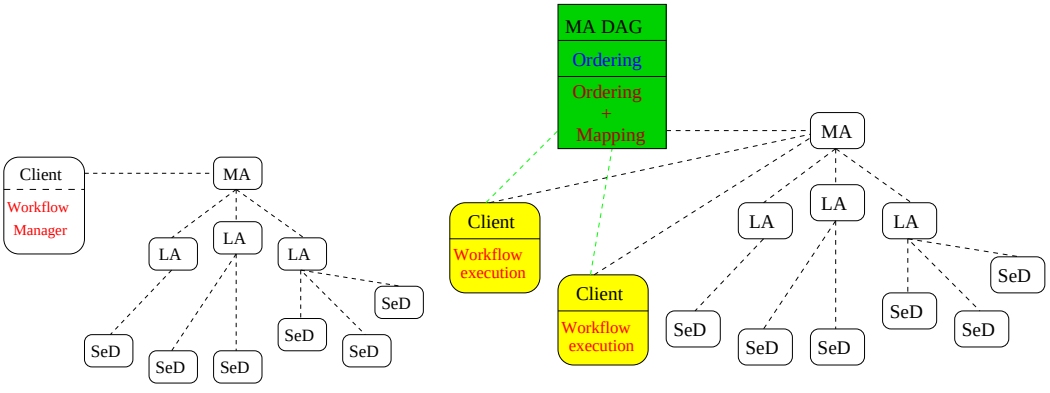


Figure 5: Software architecture of DIET workflow engine.

The two architectures presented in the previous figure can be used within the same DIET platform. The use of the MA_{DAG} is based on the user choice to use his own scheduling strategy or to use the global one provided by the MA_{DAG}. It is obvious that when the user decides not to use the MA_{DAG}, there is no collaboration between the different clients but he can use and test easily a new scheduling algorithm by plugging it in the client code. On the other hand, when the MA_{DAG} is used, the workflow submissions go through this special

agent and the multi-workflow can be handled more efficiently using core heuristics. To avoid overloading due to multiple workflow submissions from different clients, the MA_{DAG} is not responsible for workflow execution but it only manages the scheduling phase. Two working modes can be used in the MA_{DAG}: in the first mode, a complete schedule (which assigns priority and mapping for each task) is provided to the client, while in the second only task priorities are returned to the client.

3.3 Future Directions

In our future work we plan to improve the flexibility of the plug-in schedulers, improve the performance evaluation feature, port new applications, and finally test several DIET platforms at a large scale within the Grid'5000 project [13]. The transparent submission to more batch schedulers will also be supported, with the help of the work that has been performed within the Open Grid Forum DRMAA working group. In this context co-scheduling algorithms should be designed. Eventually, we could design a Service Oriented Architecture (SOA) based on DIET and benefiting from plug-in scheduler.

Concerning the data management we developed a new tool called DAGDA (Data Arrangement for Grid and Distributed Application) to support the GridRPC data management API. DAGDA is a new data manager for DIET which allows data explicit or implicit replications and advanced data management on the grid.

4 GridSolve

The purpose of GridSolve is to create the middleware necessary to provide a seamless bridge between the simple, standard programming interfaces and desktop systems that dominate the work of computational scientists and the rich supply of services supported by the emerging grid architecture. The goal is that the users of desktop systems can easily access and reap the benefits (in terms of shared processing, storage, software, data resources, etc.) of using grids. Having a broad community of scientists, engineers, research professionals and students working with the powerful and flexible tool set provided by their familiar desktop computing environment, and yet able to easily draw on the vast, shared resources of the grid for unique or exceptional resource needs, or to collaborate intensively with colleagues in other organizations and locations, is the vision that GridSolve is designed to realize.

4.1 How GridSolve Works

GridSolve is a client-agent-server (or *brokered RPC*) system which provides remote access to hardware and software resources through a variety of client interfaces.

The system consists of three entities, as illustrated in Figure 6.

- The *Client*, which needs to execute some remote procedure call. In addition to C and Fortran programs, the GridSolve client may be an interactive problem solving environment such as Matlab, Octave, or IDL (Interactive Data Language).
- The *Server* executes functions on behalf of the clients. The server hardware can range in complexity from a uniprocessor to a MPP system and the functions executed by

Figure 6: Overview of GridSolve.

the server can be arbitrarily complex. Server administrators can straightforwardly add their own function services without affecting the rest of the GridSolve system.

- The *Agent* is the focal point of the GridSolve system. It maintains a list of all available servers and performs resource selection for client requests as well as ensuring load balancing of the servers.

In practice, from the user's perspective the mechanisms employed by GridSolve make the remote procedure call fairly transparent. However, behind the scenes, a typical call to GridSolve involves several steps, as follows:

1. The client queries the agent for an appropriate server that can execute the desired function.

2. The agent returns a list of available servers, ranked in order of suitability.

3. The client attempts to contact a server from the list, starting with the first and moving down through the list. The client then sends the input data to the server.

4. Finally the server executes the function on behalf of the client and returns the results.

In addition to providing the middleware necessary to perform the brokered remote procedure call, GridSolve aims to provide mechanisms to interface with other existing grid services. This can be done by having a client that knows how to communicate with various grid services or by having servers that act as proxies to those grid services. GridSolve provides some support for the proxy server approach, while the client-side approach would be supported by the emerging GridRPC standard API [39].

4.2 Integrating User Services

We have implemented a simple technique for adding arbitrary services to a running server. First, the new service should be built as a library or object file. Then the user writes a specification of the service parameters in a gsIDL (GridSolve Interface Definition Language) file. The GridSolve problem compiler processes the gsIDL and generates a wrapper which is automatically compiled and linked with the service library or object files. Thus the services are compiled as external executables with interfaces to the server described in a standard format. The server re-examines its own configuration and installed services periodically to detect new services. In this way it becomes aware of the additional services without re-compilation or restarting of the server itself.

Normally the GridSolve server executes the actual service request itself, but in some cases it can act as a proxy to other services such as Condor. The primary benefit is that the client-to-server communication protocol is identical so the client does not need to be aware of every possible back-end service. A server proxy also allows aggregation and scheduling of resources, such as the machines in a cluster, on one GridSolve server.

4.3 Scheduling

The selection of the best server for a particular job is carried out at several layers. When a new service is added, the author should provide a rough characterization of the performance in terms of the arguments to the function. For example, sorting an N element array may be characterized with `COMPLEXITY="N * log(N)"` in the service configuration file. As the service is invoked, the server keeps track of the typical execution time for various problem sizes and uses a least squares regression to compute coefficients for an expression that more closely characterizes the expected performance. This is useful in cases where different implementations of a service have the same theoretical execution time, but very different real-world performance (*e.g.,* vendor-tuned BLAS compared with the reference BLAS). Both the theoretical and observed information are sent to the GridSolve agent, which uses them to determine the ranking of the servers. After the ranked list is returned to the client, it may choose to refine the list based on communication performance. For instance, a very fast server may not be the best choice if it is only reachable through a slow connection. Thus, the client can run a quick series of communication tests to estimate the time that it would take to send and receive the data from each of the servers. The server list is then re-sorted based on this information.

4.4 Network Address Translators

As the rapid growth of the Internet began depleting the supply of IP addresses, it became evident that some immediate action would be required to avoid complete IP address depletion. The IP Network Address Translator [22] is a short-term solution to this problem. Network Address Translation presents the same external IP address for all machines within a private subnet, allowing reuse of the same IP addresses on different subnets, thus reducing the overall need for unique IP addresses.

4.4.1 Complications in the Presence of NATs

As beneficial as NATs may be in alleviating the demand for IP addresses, they pose many significant problems to developers of distributed applications such as GridSolve [37]. Some of the problems as they pertain to GridSolve are: IP addresses may not be unique, IP address-to-host bindings may not be stable, hosts behind the NAT may not be contactable from outside, and NATs may increase connection failures.

- IP addresses are not unique – In the presence of a NAT, a given IP address may not be globally unique. Typically the addresses used behind the NAT are from one of several blocks of IP addresses reserved for use in private networks, though this is not strictly required. Consequently any system that assumes that an IP address can serve as the unique identifier for a component will encounter problems when used in conjunction with a NAT.

- IP address-to-host bindings may not be stable – This has similar consequences to the first issue in that GridSolve can no longer assume that a given IP address corresponds uniquely to a certain component. This is because, among other reasons, the NAT may change the mappings.

- Hosts behind the NAT may not be contactable from outside – This currently prevents all GridSolve components from existing behind a NAT because they must all be capable of accepting incoming connections.

- NATs may increase connection failures – Connections through NATs may sometimes be dropped spontaneously, depending on the particular NAT implementation (especially after a period of inactivity). This implies that GridSolve needs more sophisticated fault tolerance mechanisms to cope with the increased frequency of failures in a NAT environment.

To address these issues we have developed a new communications framework for GridSolve. To avoid problems related to potential duplication of IP addresses, the GridSolve components will be identified by a globally unique identifier specified by the user or generated randomly. The mapping between the component identifier and a real host will not be maintained by the GridSolve components themselves, rather there will be a discovery protocol to locate the actual machine running the GridSolve component with the given identifier. In a sense, the component identifier is a network address that is layered on top of the real network address such that a component identifier is sufficient to uniquely identify and locate

any GridSolve component, even if the real network addresses are not unique. This is somewhat similar to a machine having an IP address layered on top of its MAC address in that the protocol to obtain the MAC address corresponding to a given IP address is abstracted in a lower layer. Since NATs may introduce more frequent connection failures, we have implemented a mechanism that allows a client to submit a problem, break the connection, and reconnect later at a more convenient time to retrieve the results. We may also want to enhance the protocol to allow restarting partial transfers.

An important aspect to making this new communications model work is the *proxy*, which is a component that allows servers to exist behind a NAT. Since a server cannot accept unsolicited connections from outside the private network, it must first register with a proxy. The proxy acts on behalf of the component behind the NAT by establishing connections with other components or by accepting incoming connections. The component behind the NAT keeps the connection with the proxy open as long as possible since it can only be contacted by other components while it has a control connection established with the proxy. To maintain good performance, the proxy only examines the header of the connections that it forwards and it uses a simple table-based lookup to determine where to forward each connection. Furthermore, to prevent the proxy from being abused, authentication may be required.

4.4.2 GridSolve Proxy API

The programming interface that applications use to communicate through the proxy is based on the BSD sockets API. To make it easy for developers to modify their code to be NAT-tolerant, our API mirrors the sockets API as closely as possible.

The following functions map directly to the BSD sockets API and have the same purpose. The primary difference is in the use of the `PROXY_COMPONENTADDR` instead of `struct sockaddr` because addressing is done at the *Component ID* level.

```
proxy_socket(int domain, int type, int protocol)
proxy_bind(int s, const struct sockaddr *name, int namelen)
proxy_listen(int s, int backlog)
proxy_accept(int s, struct sockaddr *addr,
  socklen_t *addrlen)
proxy_connect(int s, PROXY_COMPONENTADDR* name)
proxy_close(int fildes)
```

The new communications API has some additional functions to initialize the system, get the component's address, and get the proxy IP address and port.

```
proxy_init(char* configFile)
proxy_get_local_addr()
proxy_get_proxy_ip()
proxy_get_proxy_port()
```

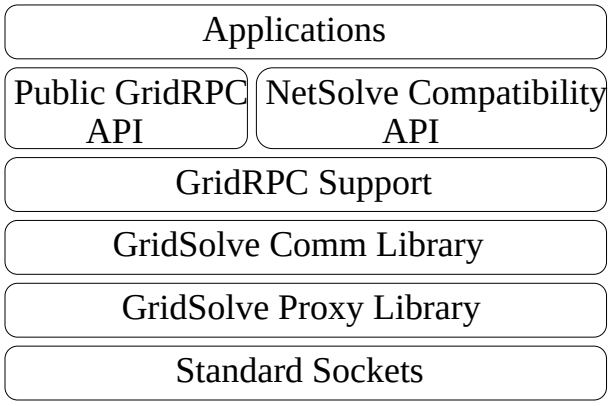


Figure 7: GridSolve GridRPC Implementation.

4.5 GridRPC Implementation

The GridRPC API specification dictates how the API itself must look, but that leaves a lot of flexibility in terms of the underlying implementation, especially in areas such as data transfer protocols, scheduling, and resource discovery.

The GridSolve implementation is composed of several layers, as illustrated in Figure 7. At the lowest layer is the standard sockets API, on top of which we have implemented our proxy library. The proxy library handles creation and manipulation of the global ID as well as interaction with the proxy server if necessary for NAT traversal. Above the proxy layer is a high-level communications library that handles connection establishment, protocols for transferring the RPC data, and data conversions such as byte order swapping and matrix transposition. Above that is a set of routines providing support for both the GridRPC public API and an optional layer for API compatibility with programs written using GridSolve. The support layer also contains some non-standard features that we are experimenting with, such as task farming and fault tolerance.

4.5.1 Delayed Function Handle Binding

The GridRPC function handle represents a mapping from a service descriptor (in this case a simple character string) to the remote server that will be used to execute the function. This mapping could be specified by the user or determined by the middleware using simple resource discovery mechanisms or possibly some more sophisticated scheduling algorithms.

The normal GridRPC calling sequence is to first initialize the handle using `grpc_function_handle_default()` followed by a call to `grpc_call()` (or one of its brethren) at some point later. In the case of the GridSolve implementation, there is a slight problem with performing the scheduling in this scenario. GridSolve relies on having access to the values of the arguments at the time the scheduling is performed so it can estimate the execution time and communication cost of sending the data. However, at the time `grpc_function_handle_default()` is called, we do not know which values will be used in the eventual call, so scheduling is not possible.

To deal with this issue, we allow the user to specify a special host name when initializing

the function handle. The special name signifies that the function handle binding should be delayed until the first time the handle is used to make a call. Subsequent calls will not cause any change in the mapping.

4.5.2 GridRPC in Interactive Environments

The various GridRPC call functions rely on a variable argument list calling sequence. While this is fine for languages like C and Fortran, it can be cumbersome when trying to link the GridRPC client with interactive environments like Matlab. An earlier version of the GridRPC specification had alternate GridRPC call functions based on an *argument stack* that could be constructed at run-time by pushing the arguments one-by-one onto the stack. The GridSolve implementation still retains these stack-based calls to ease the integration with SCEs.

4.5.3 Fault Tolerance

GridSolve was implemented with GridRPC as its primary client API, but since it was an evolution of the NetSolve project, we wanted to be able to implement a NetSolve compatibility API for supporting code written to the previous API. The NetSolve API is fairly similar, but one major difference in the call semantics is that a failed call will be automatically resent to a different server. The whole process is transparent to the user.

Thus, to support building a NetSolve API on top of our GridRPC implementation, we added fault tolerant versions of the call, probe, and wait routines. Calling probe or wait on a failed call will result in the call being performed again. These new routines are named similarly, but with the addition of the "_ft" suffix.

4.5.4 Task Farming

Another feature from the NetSolve API that we wanted to preserve for compatibility and future experimentation is *task farming*. Task farming represents an important class of distributed computing applications, where multiple independent tasks are executed to solve a particular problem. Many algorithms fit into this framework, for example, parameter-space searches, Monte-Carlo simulations and genome sequence matching.

Without using a special task farming API, a naive algorithm could be implemented by using the standard GridRPC interface and letting the GridSolve agent handle the scheduling. A user would make a series of non-blocking requests, probe to see if the requests have completed, and then wait to retrieve the results from completed requests. However this leads to problems with regard to scheduling, especially if the number of tasks is much larger than the number of servers. Alternatively, the user could try to handle the details of scheduling, but this solution requires a knowledge of the system that is not easily available to the user, and it ignores the GridSolve goal of ease-of-use.

4.5.5 Request Serialization

Normally the results of a GridRPC call must be retrieved from the same process that initiated the call, but there are several reasons a user may want to pick up the results from a

different process:

- For very long running jobs, the user may not want to tie up resources by keeping the client application running until completion. The application can be closed an restarted later to retrieve the results.
- In some cases, the user may want to initiate the job, but have the results retrieved later by a different person (or perhaps by himself, but from a different machine).
- For machines that go down regularly for various reasons (reservations, maintenance, or just plain unreliability), saving the request can provide some degree of insurance against losing results.
- As mentioned in Section 4.4, NATs can affect the stability of connections, especially very long-lived connections with no traffic, as would be likely the scenario of waiting for a long job. However, this can be rectified to a certain degree by using the asynchronous GridRPC calls, which would not normally keep an open connection.

To deal with these various scenarios, we have added request serialization and deserialization functions to our implementation. The serialization process stores into a character string all the information necessary to retrieve the results later. This string can be saved to disk and loaded into a separate process or sent to another user to be loaded into their application.

5 GridRPC System Ninf-G

5.1 Brief History of Ninf-G

Ninf-G [41, 42, 60] is a GridRPC system developed in AIST (National Institute for Advanced Industrial Science and Technology), Japan. The project started back in 1994 and the first generation implementation, called Ninf-1 [40, 52, 58] was released in 1996. Ninf-1 did not provide sufficient security capabilities such as authentication or private communication.

The second generation implementation, called **Ninf-G**, was released in 2001, which was based on Globus Toolkit 2 [47]. Thanks to Globus Toolkit, it enjoyed PKI based authentication and authorization along with private communication. As the Globus Toolkit moved on to Web Service based version 4, Ninf-G kept up with it; the ver. 4 was released Feb. 2006, and can work with several grid middleware other than Globus toolkit.

5.2 The Design of Ninf-G

Ninf-G is designed the followings in mind.

- Simplicity.
  Ninf-G is designed to be a thin layer that just does RPC. For example, it does not provide any scheduling capability by itself. Instead, it is designed so that it is easy to implement scheduling module on it. This is because scheduling strategies deeply depend on the applications and no single scheduling mechanism full-fill requirements of the application.

- Leverage existing middleware.
  To avoid duplicated effort, we decided not to implement our own proprietary protocols. Instead, we leverage existing grid middleware, such as Globus Toolkit, as much as possible. This policy allows users to utilize de-fact standard grid infrastructure, such as GridFTP servers, as part of their application written in Ninf-G.

5.2.1 Language Bindings

For the client side, Ninf-G provides client libraries written in C, which can be used also from C++ and Fortran through wrapping functions, and Java. For the server side remote library, Ninf-G supports C, C++, and Fortran. Ninf-G also supports server side MPI, meaning that numerical libraries written using MPI, such as ScaLAPACK, can be called via network using Ninf-G.

5.3 Basic Architecture of Ninf-G

A grid application constructed with Ninf-G is composed of a client program, written with the GridRPC API, and server side remote library executable modules. Ninf-G expects following three services for the underlying grid middleware.

- Authenticated invocation of remote library module
- Secure communication between the client programs and the remote library modules
- Information management for remote library interface information and remote library invocation

For authenticated invocation, Ninf-G is able to use several grid middleware, as described in detail in Section 5.6. For secure communication, Ninf-G uses Globus-IO, which is provided as a part of Globus Toolkit and enables authenticated and private communication. As information services, Ninf-G supports Globus MDS2 and MDS4, while also allows users to just use files on the local site as a information source.

The diagram shown in Figure 8 describes the overview of the Ninf-G system.

5.4 How to “Gridify” Libraries

In order to “gridify” a library, the Ninf library provider describes the interface of the library function using the Ninf IDL to publish his library function, which are only manifested and handled at the server side. The Ninf IDL supports datatypes mainly tailored for serving numerical applications: for example, the basic datatypes are largely scalars and their multi-dimensional arrays. On the other hand, there are special provisions such as support for expressions involving input arguments to compute array sizes, designation of temporary array arguments that need to be allocated on the server side but not transferred, etc.

This allows direct “gridifying’ of existing libraries that assumes array arguments to be passed by call-by-reference (thus requiring shared-memory support across nodes via software), and supplementing the information lacking in the C and Fortran typesystems regarding array sizes, array stride usage, array sections, etc.

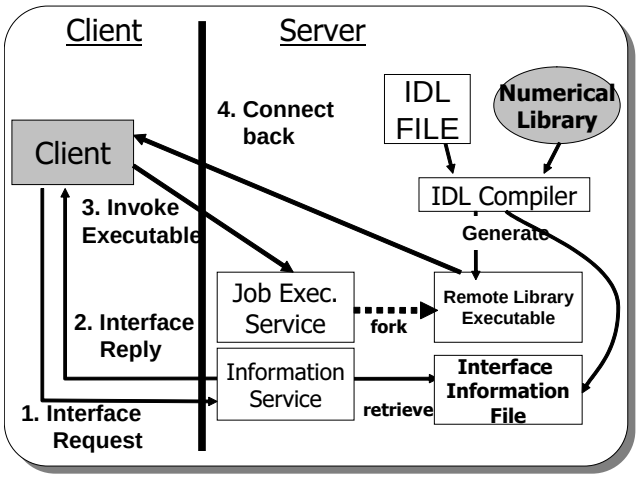


Figure 8: Ninf-G overview.

```
Module sample;
Define mmul(IN  int    N,
            IN  double A[N*N],
            IN  double B[N*N],
            OUT double C[N*N])
Required "mmul_lib.o"
Calls "C" mmul(N, A, B, C);
```

Figure 9: An example of Ninf IDL file.

As an example, interface description for the matrix multiply is shown in Figure 9, where the access specifiers `IN` and `OUT` specify whether the argument is read or written within the gridified library. Other `IN` arguments can specify array sizes, strides, etc., with size expressions. In this example, the value of N is referenced to calculate the size of the array arguments A, B, C. In addition to the interface definition of the library function, the IDL description contains the information needed to compile and link the necessary libraries. Ninf-G tools allow the IDL files to be compiled into stub main routines and makefiles, which automates compilation, linkage and registration of gridified executables.

5.5 Advanced Features of Ninf-G

Although the primal API of Ninf-G is the GridRPC standard API, Ninf-G also supports non-standard API functions to support advanced features.

5.5.1 Server Side Persistent State

One of the outstanding features is “object handle” that enables to keep persistent state on the server side. This is quite effective to reduce communication between client and server.

Assume that you have a parameter survey-type application, that requires certain amount of data except for the parameter itself for calculation. If we just use the basic GridRPC mechanism, we have to transfer the data again and again along with the parameter. With the

```
Module matmul_object;

DefClass matmul
Required "matmul.o"
{
  DefMethod setArray(IN int N, IN double A[N][N])
  "set persistent array"
  {
  extern void setArray(int, double *);
  setArray(N, A);
  }

  DefMethod multiply(IN int N, IN double B[N][N], OUT double C[N][N])
  {
  extern void multiply(int, double *, double *);
  multiply(N, B, C);
  }
}
```

Figure 10: An example of Ninf Object IDL.

```
/* create object handle */
grpc_object_handle_init_np(&handle,
                           "server.example.org",
                           "matmul_object/matmul");

/* set the left operand */
grpc_invoke_np(&handle, "setArray", N, A);

/* multiply several times reusing the left operand */
for (i = 0; i < M; i++) {
  grpc_invoke_np(&handle, "multiply", N, &B[i], &C[i]);
}
```

Figure 11: Client code fragment that uses object handle.

persistent state capability, we can stage the data in advance only once, and reuse them for successive calculations without transfer them. This will drastically reduce the data transfer amount and raise the total performance of the application.

Let us see the IDL example that defines “remote object”. In the example shown in Figure 10, we defined a object that multiplies matrices. We assume that the user wants to keep the left hand side matrix same for all the computation, therefore make the left matrix persistent. This object defines two methods, one is the `setArray` that is to transfer the left hand matrix in advance, and the other is the `multiply` that does the computation.

To utilize the remote object, the client has to use a new type called `grpc_object_handle_t` and a series of functions begins with `grpc_invoke`. A program fragment is shown in Figure 11.

5.5.2 Callback Function

Another outstanding capability is the *callback function*. The callback function is the capability to call functions on the client from the server side remote library modules. This capability allows users to visualize intermediate result on the client display and to steer computation based on the intermediate result. Another possible application of this capability is branch and bound methods where this capability is useful to broadcast intermediate result as soon as possible while keeping the grain size of the computation.

```
/* global */
int executableStatus;
int clientStatus;

void callback_func(int c[], int d[])
{
    executableStatus = c[0];
    d[0] = clientStatus;
}

main()
{
    grpc_function_handle_t handle;
    grpc_error_t result;
    int b;
    ...
    result = grpc_function_handle_init(&handle,
        "server.example.org", "test/callback_test");
    result = grpc_call(&handle, 100, &b, callback_func);
    ...
}
```

Figure 12: Client code that uses a callback function.

```
Module test;

Define callback_test(IN int a, OUT int *b,
                     callback_func(IN int c[1], OUT int d[1]))
{
    int executableStatus, clientStatus;
    executableStatus = calc(a, b);
    callback_func(executableStatus, &clientStatus);
    if (d == 1) {
        /* client is alive */
    }
}
```

Figure 13: Example of server-side IDL with a callback function.

A client program example is shown in Figure 12. The program defines a function (`callback_func`) and passes the pointer to the GridRPC API function. The corresponding remote library IDL is shown in Figure 13. Ninf-G IDL compiler generates stub functions for each callback function and passes pointers for them to remote library codes. The remote libraries can just use them as if they are ordinary function pointers.

5.5.3 Bulk Function Handle Initialization

Parameter survey type applications often requires to initialize a lot of, from tens to hundreds, function handles on a site. With serialized initialization, it takes substantial time just to initialize handles, due to the large network latency in the grid environment and slow response of the grid middleware. Fortunately, most back-end queueing systems and grid middleware support bulk job submission capability. Ninf-G provides an API function to leverage such capability to avoid the serialized initialization. Figure 15 shows code fragments for serialized and bulk handle initialization.

5.5.4 Error Detection with Timeout

In the grid environment, error detection is much more difficult than with an environment in a single site. Even with TCP connection, we cannot expect a program to detect a network failure immediately, making timeout based error-detection quiet important. Ninf-G provides

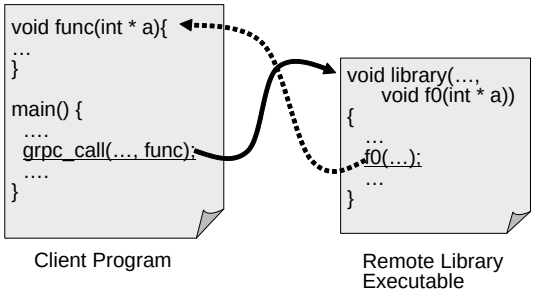


Figure 14: Callback function.

```
// serialized initialization
for (i = 0; i < n; i++)
   grpc_function_handle_init(&handles[i], host, func_name);
```

```
// bulk initialization
grpc_function_handle_array_init_np(handles, n, host, func_name);
```

Figure 15: Bulk function handle initialization.

three kinds of timeout mechanisms: invocation timeout, execution timeout, and heartbeat timeout. The first mechanism detects timeout for server process invocation. If a remote program is not activated within a time specified by the user, Ninf-G returns an error to the client program. Execution timeout detects an excessive execution time. The last mechanism is used to detect the degradation of network performance. When the heartbeat timeout is set, the library embedded in the remote library executable sends keep-alive messages to the client periodically. If the client does not receive the message over a specified period, Ninf-G returns an error. All timeout mechanisms can be used by setting attributes such as `job_maxwallTime` in the configuration file. We set these attributes and implement error check routines for all the Ninf-G functions in the scheduling code. This capability was proved to be essential in the wide area experiment described in the section 6.5.

5.6 Multiple Invocation Method

Recently, there are several grids in production. Unfortunately, the middleware used there are not standardized yet, despite the tremendous effort in the Open Grid Forum. To utilize grids out there, job invocation methods for each grid middleware are required. Ninf-G supports several grid middleware, including Globus GRAM2 [17], GRAM4 [26], Unicore [49], Condor [16], NAREGi middleware [36], and SSH, to give users the chance to utilize grids as much as possible. A Ninf-G client program can utilize all of them simultaneously.

The module actually manages job invocation, which is called *invoke server* and runs as a separate process. This is not only to avoid the possible problems on linkage with libraries provided by several grid middleware, but also to allow to use most proper languages for the target grid middleware.

The diagram shown in Figure 16 denotes the module configuration within the invoke

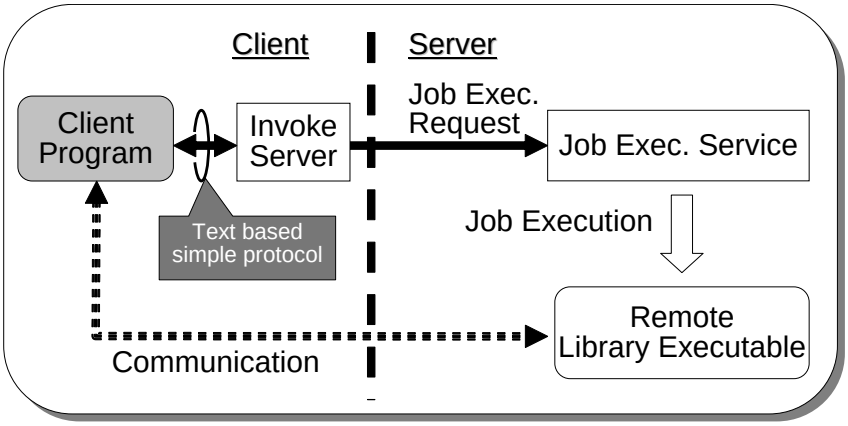


Figure 16: Invoke Server concept.

Table 1: List of Invoke Servers

Target Invoke Method	Implementation Language
Globus GRAM2	C
Globus GRAM4	python
Unicore	Java
NAREGI Middleware	Java
Condor	Java
SSH	C

server. The invoke server and the client program communicate with each other using a text-based simple protocol, which is inspired by the GAHP [48] protocol used for remote job control in Condor-G [66].

The protocol uses two streams: one is for synchronous communication and the other is for asynchronous notification from the invoke server. The former is mapped on to the standard in/out of the invoke server, while the latter is on the standard error. Note that the invoke server does not have to open socket connection to the client, making the implementation simple and easy. The protocol is simple enough and well-documented to make it easy to implement new invoke server for given grid middleware.

In Table 1, we show the list of the invoke servers and language used to implement the module. The invoke server for Globus GRAM4 is implemented as a Python script that uses commands written in C behind the seen for job invocation and management. The invoke server for Condor, implemented in Java, also uses C written command-line commands. The invoke server for SSH is provided to make it easy to try the Ninf-G capability without fully deproying the Globus Toolkit. It uses SSH to communicate with the remote server. Adding to the default “fork” method for invocation, it also supports to submit jobs using server side job queueing systems, including Sun Grid Engine [57] and Condor [16].

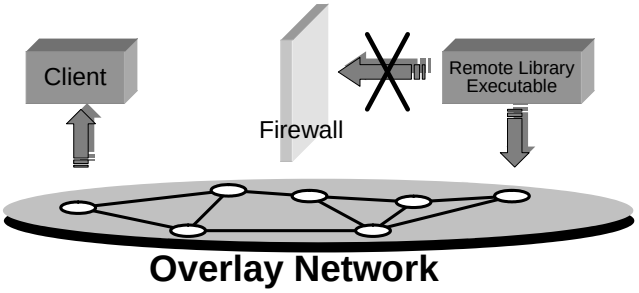


Figure 17: Ninf-G 5 design concept.

5.7 Future Direction

Network asymmetry is always the problem with distributed grid applications. Ninf-G version 4 assumes that the remote library can directly connect back to the client, which means that the remote library executable can be inside a NAT enabled private network, while the client cannot. To cope with this problem there are lot of efforts, generally called *overlay networks*, which enables symmetric communication on the asymmetric physical environment. The natural way for us is to modify the Ninf-G so that it can leverage such efforts for communication between client and remote library executable. The next generation, called Ninf-G ver. 5, which is scheduled to be released in 2008 Spring, will have generic interface to communicate with the existing overlay networks, as shown in Figure 17, and our own rather primitive overlay network implementation.

6 Applications

6.1 Cosmological simulations

RAMSES [4] application is a typical computational intensive application used by astrophysicists to study the formation of galaxies. RAMSES is used, among other things, to simulate the evolution of a collisionless, self-gravitating fluid called “dark matter” through cosmic time (see Figure 18). Individual trajectories of macro-particles are integrated using a state-of-the-art “N body solver”, coupled to a finite volume Euler solver, based on the Adaptive Mesh Refinement technics. The computational space is decomposed among the available processors using a *mesh partitionning* strategy based on the Peano–Hilbert cell ordering [64, 65].

Cosmological simulations are usually divided into two main categories. Large scale periodic boxes (see Figure 18) requiring massively parallel computers are performed on very long elapsed time (usually several months). The second category stands for much faster small scale “zoom simulations”. One of the particularity of the HORIZON[3] project is that it allows the re-simulation of some areas of interest for astronomers.

For example in Figure 19, a supercluster of galaxies has been chosen to be re-simulated at a higher resolution (highest number of particules) taking the initial information and the boundary conditions from the larger box (of lower resolution). This is the latter category we

[4] `http://www.projet-horizon.fr/`

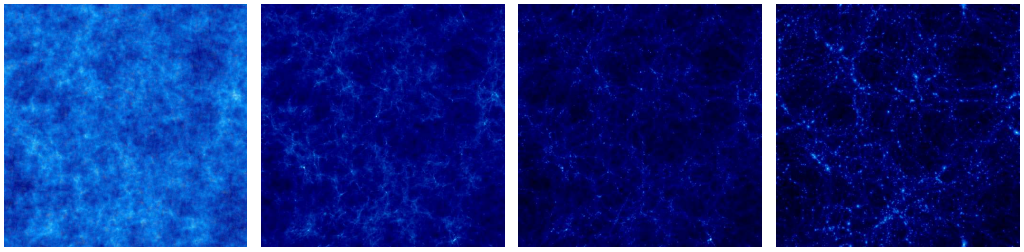


Figure 18: Time sequence (from left to right) of the projected density field in a cosmological simulation (large scale periodic box).

are interested in. Performing a zoom simulation requires two steps: the first step consists of using RAMSES on a low resolution set of initial conditions *i.e.*, with a small number of particles) to obtain at the end of the simulation a catalog of "dark matter halos", seen in Figure 18 as high-density peaks, containing each halo position, mass and velocity. A small region is selected around each halo of the catalog, for which we can start the second step of the "zoom" method. This idea is to resimulate this specific halo at a much better resolution. For that, we add in the Lagrangian volume of the chosen halo a lot more particles, in order to obtain more accurate results. Similar "zoom simulations" are performed in parallel for each entry of the halo catalog and represent the main resource consuming part of the project.

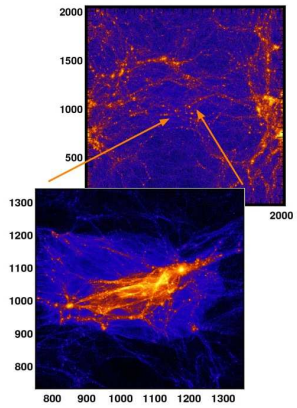


Figure 19: Re-simulation on a supercluster of galaxies to increase the resolution

RAMSES simulations are started from specific initial conditions, containing the initial particle masses, positions and velocities. These initial conditions are read from Fortran binary files, generated using a modified version of the GRAFIC[5] code. This application generates Gaussian random fields at different resolution levels, consistent with current observational data obtained by the WMAP[6] satellite observing the cosmic microwave background radiation. Two types of initial conditions can be generated with GRAFIC:

[5] `http://web.mit.edu/edbert`
[6] `http://map.gsfc.nasa.gov`

- single level: this is the “standard” way of generating initial conditions. The resulting files are used to perform the first, low-resolution simulation, from which the halo catalog is extracted.

- multiple levels: this initial conditions are used for the “zoom simulation”. The resulting files consist of multiple, nested boxes of smaller and smaller dimensions, as for Russian dolls. The smallest box is centered around the halo region, for which we have locally a very high accuracy thanks to a much larger number of particles.

The result of the simulation is a set of “snaphots”. Given a list of time steps (or expansion factor), RAMSES outputs the current state of the universe (*i.e.*, the different parameters of each particules) in Fortran binary files.

These files need post-processing with GALICS modules: HaloMaker, TreeMaker and GalaxyMaker. These three modules are meant to be used sequentially, each of them producing different kinds of information: HaloMaker detects dark matter halos present in RAMSES output files, and creates a catalog of halos. TreeMaker gives the catalog of halos, TreeMaker builds a merger tree: it follows the position, the mass, the velocity of the different particules present in the halos through cosmic time. GalaxyMaker Application applies a semi-analytical model to the results of TreeMaker to form galaxies, and creates a catalog of galaxies. Figure 20 shows the sequence of modules used to realise a whole simulation.

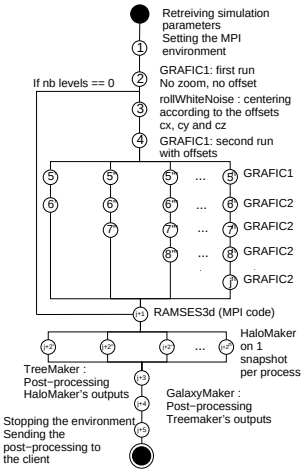


Figure 20: Workflow of a simulation

An experiment has been realized to test the scalability of DIET. This experiment has been realized on Grid’5000 [13], and the application was about cosmological computations. For this experiment, the entire grid of Grid’5000 was reserved (i.e., *among the uncrashed nodes*) which corresponded to 12 clusters that have been used on 7 sites for a duration time of 48 hours. Finally 979 machines were used with an user-defined environment containing all the needed modules for the experiment.

6.2 Environmental Modeling

A tremendous amount of planning goes into an undertaking as large as restoring the Everglades. Studies must first be done to determine what areas need to be restored and how best to do so without further damaging an already delicate ecosystem. To aid in this planning, a group at the University of Tennessee led by Dr. Lou Gross has collaborated on the development of a suite of environmental models called ATLSS (Across Tropic Level System Simulation) [25]. These models provide comparisons of the effects of alternative future hydrologic plans on various components of the biota.

This package has proven itself quite useful in the planning efforts, however it requires extensive computational facilities that are typically not available to the many stakeholders (including several federal and state agencies) involved in the evaluation of plans for restoration that are estimated to cost $8 billion. To allow greater access and use of computational models in the South Florida stakeholder community, a grid-enabled interface to the ATLSS models has been developed and is currently being used on SInRG resources. This interface provides for the distribution of model runs to heterogeneous grid nodes. The interface utilizes GridSolve for model run management and the LoRs (Logistical Runtime System) [9] toolkit and library for data and file movement. Integration of the grid interface with a web based launcher and database provides a single interface for accessing, running, and retrieving data from the variety of different models that make up ATLSS, as well as from a variety of different planning scenarios.

ATLSS, in conjunction with GridSolve and LoRS, is the first package we are aware of that provides transparent access for natural resource managers through a computational grid to state-of-the-art models. The interface allows users to choose particular models and parameterize them as the stakeholder deems appropriate, thus allowing them the flexibility to focus the models on particular species, conditions or spatial domains they wish to emphasize. The results can then be viewed within a separate GIS tool developed for this purpose.

6.3 Statistical Parametric Mapping

Statistical Parametric Mapping (SPM) is a widely used medical imaging software package. The SPM web site [56] describes the technique as follows.

> Statistical Parametric Mapping refers to the construction and assessment of spatially extended statistical process used to test hypotheses about [neuro]imaging data from SPECT/PET & fMRI. These ideas have been instantiated in software that is called SPM.

Although SPM has achieved widespread usage, little work has been done to optimize the package for better performance. In particular, little effort has gone into taking advantage of the largely parallel nature of many parts of the SPM package.

Through continuing research by Dr. Jens Gregor and Dr. Michael Thomason at the University of Tennessee, preliminary work has been done to enhance the SPM package to utilize grid resources available through GridSolve and IBP by way of the GridSolve-to-IBP library. GridSolve-to-IBP is a library built on top of LoRS and ROLFS (Read-Only Logistical File System) that allows the sharing of files between the GridSolve client and

server processes, using IBP repositories as intermediate storage. This allows programs that need access to a common set of files (*e.g.,* SPM) to export some of their functionality to a GridSolve server without having to use a shared filesystem, such as NFS.

The grid-enabled version of SPM is still under development, but executions of the preliminary version have been timed to run in one half to one third the time of unmodified code in some simulations. Once completed, the SPM software will be distributed across the SInRG resources for use in medical research, providing doctors and researchers a faster time to completion, something often critical in medical imaging analysis.

6.4 Vertex Cover and Clique Problems

A widely-known and studied problem in computer science and other disciplines is the Vertex Cover problem, which asks the following question.

> Given a graph G=(V,E) and an integer k, does G contain a set S with k or fewer vertices that covers all of the edges in G, where an edge is said to be covered if at least one of its endpoints are contained in S?

Vertex Cover is NP-complete in general, but solvable in polynomial time when k is fixed. The applications for this problem are far-reaching, including applications in bioinformatics, such as phylogeny, motif discovery, and DNA microarray analysis. The problem, however, is inherently difficult and time-consuming to solve, so efficient software packages for solving Vertex Cover are very desirable.

Research conducted by Dr. Michael Langston of the University of Tennessee aims to create an efficient software package for solving Vertex Cover. Dr. Langston and his student researchers are interested mainly in the duality between the Vertex Cover problem and the Clique problem. The Clique problem asks the following question.

> Given a graph G=(V,E) and an integer k, does G contain a set S of k nodes such that there is an edge between every two nodes in the clique?

By exploiting the duality between these two problems, they have been able to solve extremely large instances of Clique (graphs containing greater than 104 vertices). To achieve reasonable times to solution, Dr. Langston's team has developed a parallel version of their software, which is being actively run on SInRG (Scalable Intracampus Research Grid) [55] resources. The team has taken several approaches to making their application grid-aware, ranging from developing a custom scheduler and starting jobs via Secure Shell (SSH) to using popular grid middleware, such as Condor. The team has implemented a prototype version of their software that uses GridSolve to efficiently access a large number of computational resources.

6.5 Hybrid Computation with GridRPC and MPI

Although MPI has long history as a de-fact standard for parallel HPC applications, it is not suitable for writing grid application only with MPI, since it is not fault-tolerant and it requires symmetric network connectivity which is not common on the real grid.

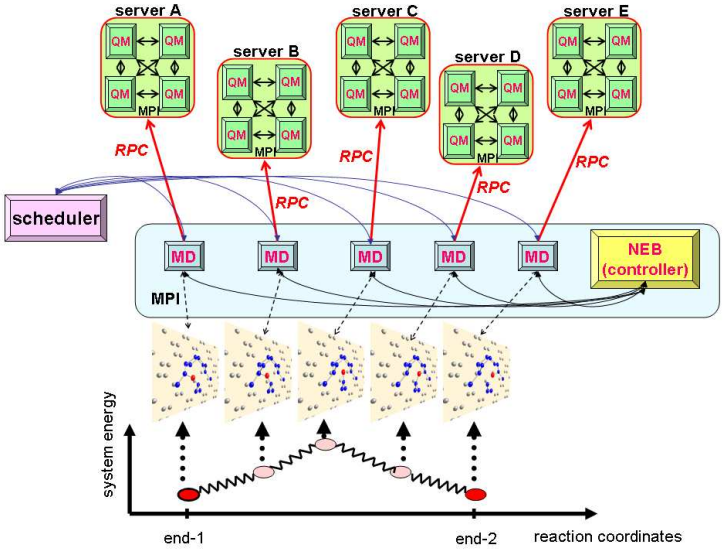


Figure 21: GridRPC and MPI hybrid approach used for multi-scale MD/QM simulation.

We proposed methods and strategies of the development and execution of grid-enabled applications which realize large-scale and long-run executions on the e-Infrastructure [61][59]. One of the key technical innovations is a programming method, which is a hybrid grid remote procedure call (GridRPC) + message passing interface (MPI) grid application framework to combine flexibility (adaptive resource allocation and migration), fault tolerance (automated fault recovery), and efficiency (scalable management of large computing resources). We have developed grid-enabled multiscale simulations based on the proposed programming model, and had large-scale empirical experiments as feasibility studies. As an application for the method, we employed multiscale simulation, that uses both of QM (Quantum Mechanics) and MD (Molecular Dynamics) for high-speed but precise simulation. Figure 21 shows the mapping of the QM and MD on to GridRPC and MPI.

We performed experiments with 6 sites spans Japan and US, 1129 processors in total (Table 2). Figure 22 shows the status of the execution. Blue bars indicate that the cluster was used for QM simulations. Red bars indicate that the cluster was not available. Yellow bars indicate that the cluster was available, however there was some limitations such as the number of available nodes was reduced due to some problems. The simulation was started using AIST P32 and F32 clusters. After 10 hours, TeraGrid clusters became available and it was automatically detected by the scheduler, which has a time table of the available clusters. The experimental results showed that the proposed programming paradigm is a promising approach for realizing sustainable grid supercomputing for large-scale scientific applications on the e-Infrastructure.

6.6 GridFMO - Quantum Chemistry of Proteins on the Grid

Another outstanding application of Ninf-G is the *GridFMO*. which is developed by recoining the Fragment Molecular Orbital (FMO) method of GAMESS with grid technology [31].

Table 2: Computing resources used for the simulations.

Cluster	Site / Grid	# CPUs to be used
F32	AIST	41
F32	AIST	32 x 6 (192)
P32	AIST	32 x 8 (256)
NCSA	TeraGrid	32 x 8 (256)
SDSC	TeraGrid	32 x 4 (128)
purdue	TeraGrid	32 x 8 (256)
Total		1129

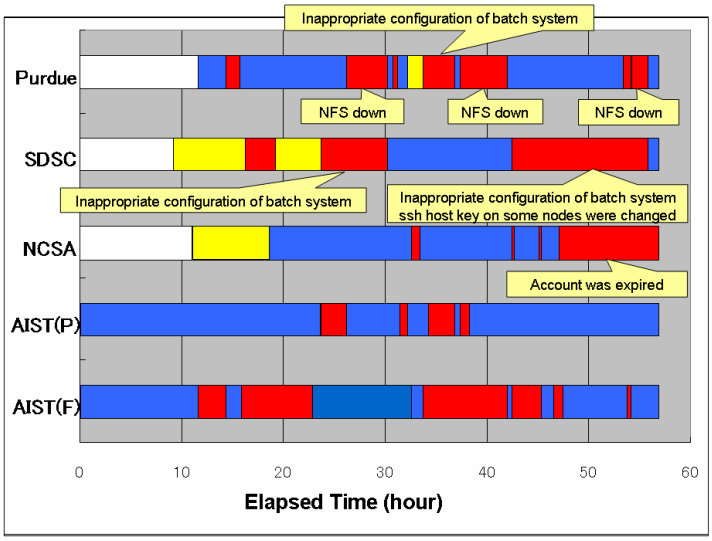


Figure 22: Simulation execution status.

With the GridFMO, quantum calculations of macro molecules become possible by using large amount of computational resources collected from many moderate-sized cluster computers.

We developed a new middleware suite on Ninf-G, whose fault tolerance and flexible resource management were found to be indispensable for long-term calculations. The middleware is composed of three objects: Bookkeeper (BK), Doorkeeper (DK), and Machine (M). The application works as a Client (C) to both BK and DK. The relationship among them is depicted in Figure 23.

The GridFMO was used to draw ab initio potential energy curves of a protein motor system with 16,664 atoms. For the calculations, 10 cluster computers over the Pacific rim were used, sharing the resources with other users via batch queue systems on each machine. A series of 14 GridFMO calculations were conducted for 70 days, coping with more than 100 problems cropping up. The FMO curves were compared against the molecular mechanics (MM), and it was confirmed that (1) the FMO method is capable of drawing smooth curves

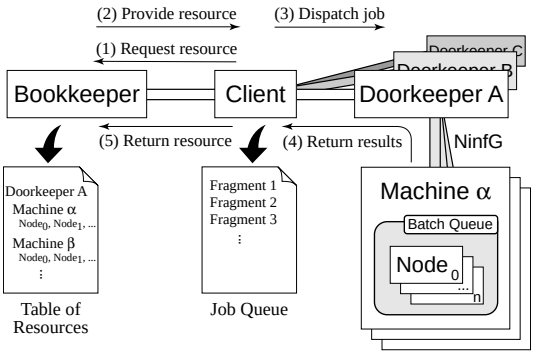


Figure 23: Structure of the middleware composed of Ninf-G.

despite several cut-off approximations, and that (2) the MM method is reliable enough for molecular modeling.

7 Conclusion and Future Work

We have presented the GridRPC model and the corresponding GridRPC API, which is now an Open Grid Forum standard. It is a simple, powerful, flexible, and effective means to execute jobs over the grid. Indeed, using the API, one can easily write a client to submit a request on any GridRPC compliant middleware, get tasks remotely launched and executed, and obtain some results. Work can even use data parallelism, task parallelism or a mixed model to achieve greater performance.

Numerous middleware are now GridRPC compliant. Here we have presented three of them which are among the most used, and whose projects actively participate to the next step of the work performed in the OGF Working Group concerning the GridRPC Data Management API, namely DIET, GridSolve, and Ninf.

Each middleware has been designed and implemented to tackle some special problematics. Hence, DIET extends the GridRPC model and relies on a hierarchy of agents where communications are addressed with a CORBA layer. DIET focus on deployment (mapping of DIET components for platform load-average) and scheduling issues, which is distributed in the hierarchy and can be application specific; GridSolve relies on sockets and addresses at the same time security issues with the encryption of communications and deployment issues to be able to maintain communications for example behind NAT; Ninf has integrated the use of Globus components to launch and manage jobs. It proposes for example some authentication methods which are necessary to use some resources distributed in Japan. Examples of use of these middlewares have been described. They show a Physics application with cosmological problems, medical issues, and chemistry application integrated as services on a grid platform which can be accessed via a web page executing a small GridRPC client.

The next steps evolution of GridRPC can be foreseen. First of all, as applications require more and more amount of data, the main focus will be on the GridRPC Data Management. Once the Data Management API is done, implementations with the use of different data will

lead to address the interoperability of the different implementations. From this point, there are two main directions to be followed. First, the conception of a fully interoperable client submitting to any middleware and using the Data Management API to address performance in the problem resolution should answer to the OGF Working Group existence. Second, the GridRPC API may be extended to converge with Web Services in the Service Oriented Architecture paradigm.

Acknowledgements

We would like to thanks several people who have contributed to the success of the GridRPC API and our middleware developments: Henri Casanova, Jack Dongarra, Craig Lee, Satoshi Matsuoka, Andre Merzky, Jean-Marc Nicod, Laurent Philippe, and each person involved in DIET, Ninf, and GridSolve development.

DIET was developed with financial support from the French Ministry of Research (RNTL GASP and ACI ASP) and the ANR (Agence Nationale de la Recherche) through the LEGOproject referenced ANR-05-CIGC-11 and the Gwendia project referenced ANR-06-MDCA-009.

References

[1] Y. Aida, Y. Nakajima, M. Sato, T. Sakurai, D. Takahashi, and T. Boku. Performance Improvement by Data Management Layer in a Grid RPC System. In Yeh-Ching Chung and José E. Moreira, editors, *GPC. Advances in Grid and Pervasive Computing, First International Conference, GPC 2006, Taichung, Taiwan, May 3-5, 2006, Proceedings*, volume 3947 of *Lecture Notes in Computer Science*, pages 324–335. Springer, 2006.

[2] K. Amin, G. von Laszewski, M. Hategan, N.J. Zaluzec, S. Hampton, and A. Rossi. GridAnt: A Client-Controllable Grid Workflow System. In *HICSS*, volume 07:70210c, 2004.

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

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

[5] D. Bachmann and A. Goller. An Image Processing Backend Based on Java and CORBA. In *Earth Observation & Geospatial Web and Internet Workshop*, February 1999.

[6] C.K. Baru, R.W. Moore, A. Rajasekar, and M. Wan. The SDSC Storage Resource Broker. In Stephen A. MacKay and J. Howard Johnson, editors, *CASCON. Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative Research*, page 5. IBM, December 1998.

[7] F. Baude, D. Caromel, and M. Morel. From Distributed Objects to Hierarchical Grid Components. In *International Symposium on Distributed Objects and Applications (DOA)*, volume 2888 of *Lecture Notes in Computer Science*, pages 1226–1242, Catania, Sicily, Italy, November 2003. Springer Verlag.

[8] M. Beck, D. Arnold, A. Bassi, F. Berman, H. Casanova, J. Dongarra, T. Moore, G. Obertelli, J. Plank, and M. Swany. Middleware for the Use of Storage in Communication. *Parallel Computing*, 28(12):1773–1787, December 2002.

[9] M. Beck, Y. Ding, S. Atchley, and J. S. Plank. Algorithms for High Performance, Wide-area Distributed File Downloads. *Parallel Processing Letters*, 13(2):207–224, June 2003.

[10] F. Berman, G.C. Fox, and A.J.H. Hey, editors. *Grid Computing: Making the Global Infrastructure a Reality*. Wiley, 2003.

[11] G. Bosilca, G. Fedak, and F. Cappello. OVM: Out-of-order Execution Parallel Virtual Machine. *Future Generation Computer Systems*, 18:525–537, 2002.

[12] M. Brzezniak and N. Meyer. Optimisation of the Usage of Mathematical Libraries in the Grid Environment. In *Proceedings of the Second Cracow Grid Workshop*, pages 74–86, Cracow, Poland, 2002.

[13] F. Cappello, E. Caron, M. Dayde, F. Desprez, E. Jeannot, Y. Jegou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, and O. Richard. GRID'5000: A Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform. In *SC'05: Proc. The 6th IEEE/ACM International Workshop on Grid Computing Grid'2005*, pages pages 99–106, Seattle, USA, November 2005.

[14] F. Cappello, S. Djilalia, G. Fedak, T. Herault, F. Magniette, V. Néri, and O. Lodygensky. Computing on large-scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid. *Future Generation Computer Systems*, 21(3):417–437, March 2005.

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

[16] Condor. http://www.cs.wisc.edu/condor/.

[17] K. Czajkowski, I. Foster, C. Kesselman, N. Karonis, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metacomputing Systems. In *Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[18] H. Dail and F. Desprez. Experiences with Hierarchical Request Flow Management for Network-Enabled Server Environments. In *International Journal of High Performance Computing Applications*, volume 20(1), pages 143–157, 2006.

[19] B. Del-Fabbro, D. Laiymani, J.-M. Nicod, and L. Philippe. A Data Persistency Approach for the DIET Metacomputing Environment. In Hamid R. Arabnia, Olaf Droegehorn, and S. Chatterjee, editors, *International Conference on Internet Computing*, pages 701–707, Las Vegas, USA, June 2004. CSREA Press.

[20] DIET: Distributed Interactive Engineering Toolbox. `http://graal.ens-lyon.fr/DIET`.

[21] S. Djilali, T. Herault, O. Lodygensky, T. Morlier, G. Fedak, and F. Cappello. RPC-V: Toward Fault-Tolerant RPC for Internet Connected Desktop Grids with Volatile Nodes. In *Proceedings of the ACM/IEEE Supercomputing'2004 Conference*, November 2004.

[22] K. Egevang and P. Francis. The IP Network Address Translator (NAT). RFC 1631, University of Utah, Department of Mathematics, May 1994.

[23] T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.L. Truong, A. Villazon, and M. Wieczorek. ASKALON: A Grid Application Development and Computing Environment. In *GRID '05: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 122–131, Washington, DC, USA, 2005. IEEE Computer Society.

[24] M.C. Ferris, M.P. Mesnier, and J.J. Moré. NEOS and Condor: Solving Optimization Problems Over the Internet. *ACM Transactions on Mathematical Software*, 26(1):1–18, March 2000.

[25] D.M. Fleming, D.L. DeAngelis, L.J. Gross, R.E. Ulanowicz, W.F. Wolff, W.F. Loftus, and M.A. Huston. ATLSS: Across-Trophic-Level System Simulation for the Freshwater Wetlands of the Everglades and Big Cypress Swamp. Technical report, National Biological Service Technical Report, 1994.

[26] I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In *IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779*, pages 2–13, 2005.

[27] I. Foster and C. Kesselman, editors. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2004.

[28] M. Good and J.-P. Goux. iMW : A Web-based Problem Solving Environment for Grid Computing Applications. Technical report, Department of Electrical and Computer Engineering, Northwestern University, 2000.

[29] GrADS. `http://www.hipersoft.rice.edu/grads/`.

[30] T. Haupt, E. Akarsu, and G. Fox. WebFlow: A Framework for Web Based Metacomputing. *Future Generation Computer Systems*, 16(5):445–451, March 2000.

[31] T. Ikegami, J. Maki, T. Takami, Y. Tanakaa, M. Yokokawa, S. Sekiguchi, and M. Aoyagi. GridFMO - Quantum Chemistry of Proteins on the Grid. In *Proc. of Grid 2007*, 2007.

[32] N. Kapadia, J. Robertson, and J. Fortes. Interfaces Issues in Running Computer Architecture Tools via the World Wide Web. In *Workshop on Computer Architecture Education at ISCA 1998*, Barcelona, 1998. `http://www.ecn.purdue.edu/labs/punch/`.

[33] R. Keller, B. Krammer, M.S. Mueller, M.M. Resch, and E. Gabriel. MPI Development Tools and Applications for the Grid. In *Workshop on Grid Applications and Programming Tools, held in conjunction with the GGF8 meetings*, Seattle, June 2003.

[34] T. Kielmann, A. Merzky, H.E. Bal, F. Baude, D. Caromel, and F. Huet. Grid Application Programming Environments. Technical Report TR-0003, Institute on Problem Solving Environment, Tools and Grid Systems, CoreGRID - Network of Excellence, June 2005.

[35] S. Matsuoka and H. Casanova. Network-Enabled Server Systems and the Computational Grid. `http://www.eece.unm.edu/\~dbader/grid/WhitePapers/GF4-WG3-NES-whitepape%r-draft-000705.pdf`, July 2000. Grid Forum, Advanced Programming Models Working Group whitepaper (draft).

[36] S. Matsuoka, S. Shinjo, M. Aoyagi, S. Sekiguchi, H. Usami, and K. Miura. Japanese Computational Grid Research Project: NAREGI. *Proceedings of the IEEE*, 93(3):522–533, March 2005.

[37] K. Moore. Recommendations for the Design and Implementation of NAT-Tolerant Applications. Internet-draft, University of Tennessee, February 2002. `http://old.iptel.org/ietf/firewall/arch/draft-moore-nat-tolerance-recom%mendations-00.txt`.

[38] MPICH-G. `http://www.hpclab.niu.edu/mpi/`.

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

[40] H. Nakada, M. Sato, and S. Sekiguchi. Design and Implementations of Ninf: towards a Global Computing Infrastructure. *Future Generation Computing Systems, Metacomputing Issue*, 15(5-6):649–658, 1999.

[41] H. Nakada, Y. Tanaka, S. Matsuoka, and S. Sekiguchi. *Grid Computing: Making the Global Infrastructure a Reality*, chapter Ninf-G: a GridRPC system on the Globus toolkit, pages 625–638. John Wiley & Sons Ltd, March 2003.

[42] Ninf. Ninf: A Global Computing Infrastructure. http://ninf.apgrid.org/.

[43] T.M. Oinn, M. Addis, J. Ferris, D. Marvin, R.M. Greenwood, T. Carver, M.R. Pocock, A. Wipat, and P. Li. Taverna: A Tool for the Composition and Enactment of Bioinformatics Workflows. In *Bioinformatics*, volume 20(17), pages 3045–3054, November 2004.

[44] M.P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40(11):38–45, 2007.

[45] PEGASUS. `http://pegasus.isi.edu/`.

[46] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski. The Internet Backplane Protocol: Storage in the Network. In *NetStore '99: Network Storage Symposium. Internet2*, October 1999.

[47] Globus Project. The Globus Project Web Page. http://www.globus.org.

[48] Globus ASCII Helper Protocol. http://www.cs.wisc.edu/condor/gahp/.

[49] M. Romberg. The UNICORE Architecture - Seamless Access to Distributed Resources. In *Proc. of 8th IEEE International Symposium on High Performance Distributed Computing (HPDC8)*, pages 287–293, 1999.

[50] M. Sato, T. Boku, and D. Takahashi. OmniRPC: a Grid RPC System for Parallel Programming in Cluster and Grid Environment. In *3rd IEEE/ACM International Symposium on CLuster Computing and the Grid (CCGRID'03)*, pages 206–213, 2003.

[51] M. Sato, M. Hirano, Y. Tanaka, and S. Sekigushi. OmniRPC: A Grid RPC Facility for Cluster and Global Computing in OpenMP. In *WOMPAT*, volume 2104 of *Lecture Notes in Computer Science*, pages 130–136. Springer Verlag, 2001.

[52] M. Sato, H. Nakada, S. Sekiguchi, S. Matsuoka, U. Nagashima, and H. Takagi. Ninf: A Network based Information Library for a Global World-Wide Computing Infrastracture. In *Proc. of HPCN'97 (LNCS-1225)*, pages 491–502, 1997.

[53] 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.

[54] G. Singh, E. Deelman, G. Mehta, K. Vahi, M.H. i Su, G.B. Berriman, J. Good, J.C. Jacob, D.S. Katz, A. Lazzarini, K. Blackburn, and S. Koranda. The Pegasus Portal: Web Based Grid Computing. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing.*, pages 680–686, New York, NY, USA, 2005. ACM Press.

[55] Scalable Intracampus Research Grid. `http://icl.cs.utk.edu/sinrg`.

[56] Statistical Parametric Mapping. `http://www.fil.ion.ucl.ac.uk/spm/`.

[57] Sun Grid Engine. http://gridengine.sunsource.net.

[58] A. Takefusa, S. Matsuoka, H. Ogawa, H. Nakada, H. Takagi, M. Sato, S. Sekiguchi, and U. Nagashima. Multi-client LAN/WAN Performance Analysis of Ninf: a High-Performance Global Computing System. In *Supercomputing '97*, 1997.

[59] Takemiya, H. and Tanaka, Y. and Sekiguchi, S. Sustainable Adaptive Grid Supercomputing: Multiscale Simulation of Semiconductor Processing across the Pacific. In *Proc. of SC06*, 2006.

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

[61] Y. Tanaka, H. Takemiya, Y. Song, S. Sekiguchi, S. Ogata, T. Kouno, R.K. Kalia, A. Nakano, and P. Vashishta. Implementation and evaluation of sustainable multiscale simulations on the grid. In *Proc. of Third Asia-Pacific Congress on Computational Mechanics*, 2007.

[62] Y. Tanimura, K. Seymour, E. Caron, A. Amar, H. Nakada, Y. Tanaka, and F. Desprez. Interoperability Testing for The GridRPC API Specification. In *GFD-E.102, GridRPC Working Group*, April 2007.

[63] Condor Team. The Directed Acyclic Graph Manager.

[64] R. Teyssier. Cosmological hydrodynamics with adaptive mesh refinement. A new high resolution code called RAMSES. *Astronomy and Astrophysics*, 385:337–364, 2002.

[65] R. Teyssier, S. Fromang, and E. Dormy. Kinematic dynamos using constrained transport with high order Godunov schemes and adaptive mesh refinement. *Journal of Computational Physics*, 218:44–67, October 2006.

[66] D. Thain, T. Tannenbaum, and M. Livny. Condor and the Grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002.

[67] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed Computing in Practice: the Condor Experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[68] R.V. van Nieuwpoort, J. Masen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H.E. Bal. Ibis: a Flexible and Efficient Java-based Grid Programming Environment. *Concurrency and Computation: Practice and Experience*, 17(7-8):1079–1107, 2005.

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

[70] J. Yu and R. Buyya. A Taxonomy of Workflow Management Systems for Grid Computing, 2005.