CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
*Concurrency Computat.: Pract. Exper.* 2000; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

# Tunable Scheduling in a GridRPC Framework

A. Amar and R. Bolze and Y. Caniou and
E. Caron[*,†] and A. Chis and F. Desprez and
B. Depardon and J.-S. Gay and G. Le Mahec and
D. Loureiro

*LIP Laboratory (UMR CNRS, ENS Lyon, INRIA, UCBL 5668)*
*GRAAL Project*
*46 Allée d'Italie, F-69364 Lyon Cedex 07*

**SUMMARY**

**Among existing grid middleware approaches, one simple, powerful, and flexible approach consists of using servers available in different administrative domains through the classic client-server or Remote Procedure Call (RPC) paradigm. Network Enabled Servers (NES) implement this model also called GridRPC. Clients submit computation requests to a scheduler whose goal is to find a server available on the grid using some performance metric. The aim of this paper is to give an overview of a NES middleware developed in the GRAAL team called DIET and to describe recent developments around plugin schedulers, workflow management, and tools. DIET (Distributed Interactive Engineering Toolbox) is a hierarchical set of components used for the development of applications based on computational servers on the grid.**

KEY WORDS: Grid Computing, Scheduling, Workflow Management, Middleware Deployment, GridRPC

## 1. INTRODUCTION

Large problems ranging from numerical simulation to life science can now be solved through the Internet using grid middleware. Several approaches exist for porting applications to grid platforms; examples include classic message-passing, batch processing, web portals, and

*Correspondence to: Eddy Caron
LIP Laboratory
École normale supérieure de Lyon
46 Allée d'Italie, F-69364 Lyon Cedex 07 France.
†E-mail: Eddy.Caron@ens-lyon.fr

Contract/grant sponsor: French Ministry of Research (RNTL GASP, ACI Grid ASP, and Grid'5000)
Contract/grant sponsor: ANR project LEGO; contract/grant number: ANR-05-CIGC-11

*Received 28 February 2007*
*Revised 30 June 2007*

GridRPC systems [25]. This last approach implements a grid version of the classic Remote Procedure Call (RPC) model. Clients submit computation requests to a scheduler that locates one or more servers 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, relatively 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.

The Distributed Interactive Engineering Toolbox (Diet) [12, 15] project is focused on the development of a scalable middleware with initial efforts focused on distributing 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 the information given in the client's 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 either in a time-shared manner, either sequentially, making servers buffer some work, with a parameter we can defined number of concurrent jobs at a given moment on a given server [13].

Several other Network Enabled Server (NES) systems have been developed in the past [2, 19]. Among them, NetSolve [3], Ninf [20], and OmniRPC [24] have particularly pursued research involving the GridRPC paradigm [25]. NetSolve, developed at the University of Tennessee, Knoxville allows the connection of clients to a centralized agent and requests are sent to servers. This centralized agent maintains a list of available servers along with their capabilities. Servers report information about their status at given intervals, and scheduling is done based on simple models provided by the application developers, Linpack benchmarks executed on remote servers, and/or information given by the Network Weather Service (NWS). Some fault tolerance is also provided at the agent level. Data management is managed either through request sequencing or using the Internet Backplane Protocol (IBP). Client Proxies ensure portability and interoperability with other systems like Ninf or Globus [4]. Ninf is a NES system developed at the Grid Technology Research Center, AIST in Tsukuba. Close to NetSolve in its initial design choices, it has evolved towards several interesting approaches using either Globus [28, 31] or Web Services [26]. Fault tolerance is also provided using Condor and a checkpointing library [21]. The performance of the platform can be studied using a powerful tool called BRICKS. As compared to the NES middleware systems described above, Diet is interesting because of the use of distributed scheduling to provide better scalability, the ability to tune its scheduling behavior using several APIs, and the use of Corba as a core middleware.

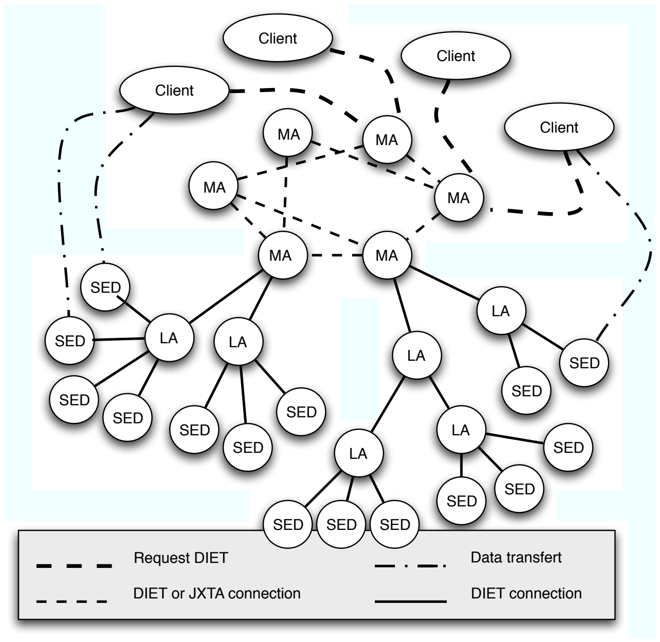


Figure 1. Diet hierarchical organization.

In this paper, we present the last developments done within the Diet project that will provide the user with an efficient, scalable for the deployment of large scale applications following the GridRPC standard over the grid. This paper is organized as follows. In Section 2, we recall the architecture of the Diet middleware and the characteristics that make it scalable over large scale grids. Then in Section 3, we describe our most recent developments in resource and server management. Core utilities for Diet management and monitoring, and the visualization of Diet's behavior on large scale platforms are described in Section 4. Finally, before a conclusion, we describe two new applications ported over Diet.

## 2. DIET ARCHITECTURE

The Diet architecture is hierarchical for 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 1 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 the agent hierarchy. The 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.

Finally, NES environments like Ninf and NetSolve use a classic socket communication layer. Nevertheless, several problems with this approach have been pointed out such as the lack of portability or limitations in the number of sockets that can be opened at once. A distributed object environment such as Corba has been proven to be a good base for building applications that manage access to distributed services. It provides transparent communication in heterogeneous networks, but it also offers a framework for the large scale deployment of distributed applications. Moreover, Corba systems provide a remote method invocation facility with a high level of transparency. This transparency should not dramatically affect the performance since the communication layers have been carefully optimized in most Corba implementations [14]. Thus, Corba has been chosen as a communication layer in Diet.

Table I. Standard estimation tags used in DIET.

| Information tag starts with EST_ | multi-value | Explanation |
|---|---|---|
| *TCOMP* | | the predicted time to solve a problem |
| *TIMESINCELASTSOLVE* | | time since last solve started (seconds) |
| *FREECPU* | | amount of free CPU (fraction between 0 and 1) |
| *LOADAVG* | | CPU load average |
| *FREEMEM* | | amount of free memory (Mb) |
| *NBCPU* | | number of available processors |
| *CPUSPEED* | x | frequency of CPUs (MHz) |
| *TOTALMEM* | | total memory size (Mb) |
| *BOGOMIPS* | x | the BogoMips |
| *CACHECPU* | x | cache size CPUs (Kb) |
| *TOTALSIZEDISK* | | size of the partition (Mb) |
| *FREESIZEDISK* | | amount of free space on partition (Mb) |
| *DISKACCESREAD* | | average time to read from disk (Mb/sec) |
| *DISKACCESWRITE* | | average time to write to disk (Mb/sec) |
| *ALLINFOS* | x | [empty] fill all possible fields |

## 3. DIET SCHEDULING

### 3.1. Plug-in Schedulers

DIET provides a special feature for scheduling 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. The performance estimation values to be used for scheduling are stored in a performance estimation vector 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 values to be stored in this structure can be provided by CoRI (Collector of Resource Information), which will be described in Section 3.2.

The standard values are to be identified based on standard estimation tags given in Table I. 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. In the performance estimation routine, the SED developer should store in the provided estimation vector any performance data to be used in the server response aggregation methods. At the time a DIET service is defined, an aggregation method - the logical mechanism by which SED responses are sorted - is associated with the service. If application-specific data are supplied (*i.e.*, the estimation function has been redefined), an alternative method for aggregation is needed.

Currently, a basic priority scheduler has been implemented, enabling an application developer to specify a series of performance values that are to be optimized in succession. From the point of view of an agent, the aggregation phase is essentially a sort of server responses from its children. A priority scheduler logically uses a series of user-specified tags to perform the pairwise server comparisons needed to construct the sorted list of server responses.

### 3.2. Collectors of Resource Information

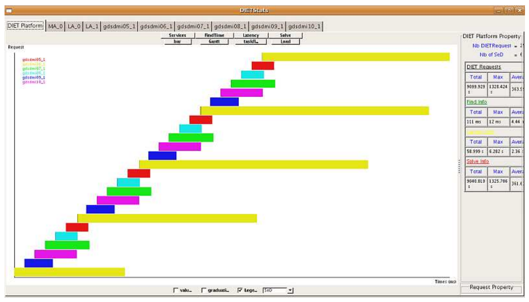
As we have seen in the previous section, to make a good decision the scheduler requires a measurement tool. In particular, DIET needs reliable resource information from grid resource information services. In this section, we introduce the requirements of DIET for a grid information service and the architecture of a new tool called Collectors of Resource Information (CoRI).

For some time DIET was dependent on a performance prediction tool called FAST [23]. Recently, we added new performance evaluation functionalities to DIET. We are now able 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. We propose a new feature which provides a basic set of performance measurements that can satisfy basic scheduler needs. 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.

The new tool has to solve two main problems. First, it must provide basic measurements that are available regardless of the execution environment. The service developer can then rely on this collector of resource information even if no other resource services like FAST or NWS are available . Secondly, the tool must manage the use of different collectors at the same time and in a similar way. We offer two solutions to these problems: the **CoRI-Easy** collector for the first problem, namely the collector, and the **CoRI Manager** for the second problem, namely management of different collectors. In general, we refer to these two solutions together as the **CoRI** tool.

CoRI-Easy is a set of simple requests for basic resource information, and CoRI Manager allows developer teams to add other resource information services. As CoRI-Easy is a resource information service, it is natural to add it as a collector in the new CoRI Manager. FAST is also available as a collector in the Manager. In addition, it is possible to add new collectors.

The CoRI Manager allows access to different **modules**, also referred to as **collectors**. A module is any kind of element that can provide information about the system. This **modularity** allows the separation of measurement sources and the selection of a module. Even if the manager should unify the different resource information services, the trace of data remains, and so the origin can be determined. For example, it could be important to distinguish the data coming from the CoRI-Easy module and the FAST module, because the information from FAST may give a more accurate estimation of the real value. The **extensibility** of the system is also ensured by its modular design. Because the interface of the manager allows the


(a) Round Robin Scheduler

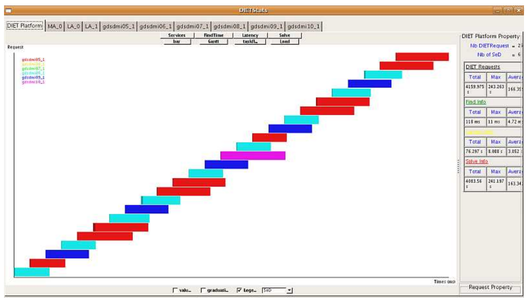

(b) CPU Scheduler

Figure 2. Comparison between the taskflows for 25 consecutive requests with task inter-arrival time equal to 1 minute.

addition of a new module in some steps, additional modules like Ganglia or NWS can easily be added.

To conclude this section, Figure 2 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.

## 3.3. DIET Batch Scheduler Management

Parallel grid resources (parallel machines or clusters of workstations) are generally managed by a reservation batch system such as Loadleveler[*], PBS[†], or OAR[‡]. 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 information 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 suit 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.

### *3.3.1. Transparently Submitting to Batch Schedulers*

Our work relies on the Elagi[*] library. It provides in particular the possibility to submit jobs to batch systems including Loadlever, Sun Grid Engine[†] and PBS. We have extended the recognized systems list with the OAR system and we plan to complete integration for the WMS system used in the EGEE[‡] project (*Enabling Grids for E-science in Europe*).

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 environment variables to perform the necessary abstraction to the site where the job is executed. For example, the generic variable `DIET_NAME_FRONTALE` is the identity of the site access point and can be used to ease data management; `DIET_BATCH_NODESFILE` is the name of the file containing the identity of the batch allocated nodes which is necessary for MPI execution, etc. The SeD program must end by a call to `diet_submit_call()`, which builds and submits the script at execution time.

To measure the performance gain involved by such a feature, Figure 3 shows the mean response time, which is the mean duration of a job in the grid system (*i.e.*, the date of the task completion minus the date of the task submission) obtained for three experiments conducted on two platforms. The two platforms are composed of one client, one agent and: (1) five SeDs, one deployed *on each* of 5 identical nodes; (2) one SeD deployed on the access point of a 7 nodes parallel resource: it can directly submit jobs to the batch scheduler. Thus, using the batch possibly leads to a gain of two computing nodes (as well as a gain in deployment). The experiment (a) is composed of a client submitting five identical tasks, each requiring 348

[*]http://www-03.ibm.com/servers/eserver/clusters/software/loadleveler.html
[†]http://www.clusterresources.com/pages/products/torque-resource-manager.php
[‡]http://oar.imag.fr/
[*]http://grail.sdsc.edu/projects/elagi/
[†]http://www.sun.qassociates.co.uk/software-grid-engine.htm
[‡]http://public.eu-egee.org/

seconds of CPU, some I/O for file transfer, at a 30 seconds rate, (b) and (c) are composed of ten identical tasks but (c) differs by a 2 seconds rate.

One can observe that for experiment (a), the mean response time (which is also the response time observed for each task in this case) is lower on the first platform. Indeed, we can see here the overhead of using batch resources: the SeD is monitoring batch jobs each 30 seconds in order not to load the parallel resource access point. Hence, the SeD is notified of the batch completion only at some monitoring events. The monitoring rate can be changed to adapt to a site behavior, but the overhead is low compared to parallel job durations in general. Furthermore, as shown in experiment (b), as soon as the grid platform is loaded (tasks are submitted at dates where no resource is idle), the mean response time (including the overhead) is lower for a parallel resource leaded by one unique SeD. As Figure 3.c describes, the gap still increases when the grid utilization is more important.

### *3.3.2. Simbatch, a Performance Prediction Module*

Simbatch[*] is a C API which relies on the Simgrid[16] grid simulator to provide models of clusters and their batch systems for multi-site grid simulations of parallel tasks. Some batch systems have already been implemented, like PBS and OAR (which respectively rely on FCFS and Conservative Backfilling), but the API is defined to easily integrate new ones.

Simbatch has been designed to fulfill numerous goals such as facilitating the conception and evaluation of grid scheduling algorithms using batch systems. Experiments have been undertaken to validate the batch simulator. Figure 4 shows representative results for an experiment composed of 100 tasks, whose input data and output data sizes are drawn uniformly between 1 and 20 Mbytes, whose computation time is drawn uniformly between 600 and 800 seconds, and where the required number of processors is between 1 and 5. The platform is composed of 7 machines, interconnected by a star topology and managed by an OAR batch system. The experiment has been run on a real architecture and simulated with the Simbatch tool. One can observe as a function of the ending date of the tasks the percentage of error between the measured flow (*i.e.*, task duration) in a real OAR batch system and the flow obtained for the same experiment with Simbatch. Note that the error is in general less than 1% in a 22 hours experiment.

The high precision of Simbatch simulation results makes it possible to use it as a performance prediction module in the Diet environment, more particularly within the SeD part for the monitoring step and during the batch submission step. Indeed, assuming that a performance prediction function for the service is given by the application programmer in the SeD server, Simbatch can simulate several scenarios to choose the best number of resources to use for the application to finish the soonest, as well as the corresponding walltime. These information can then be 1) used for scheduling in agents upward the hierarchy, 2) eventually re-calculated and used when the job is effectively submitted to the chosen parallel resource, during the automatic generation of the batch script.

[*]http://graal.ens-lyon.fr/Simbatch

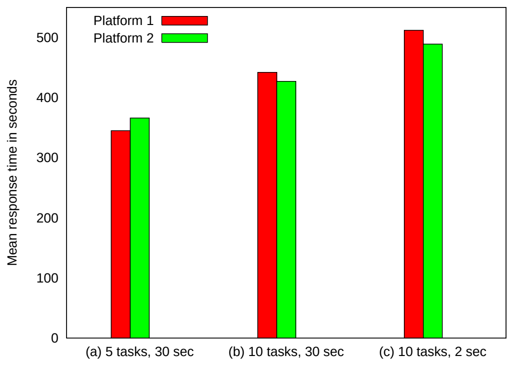


Figure 3. Mean response time for identical requests set on a two testbeds.

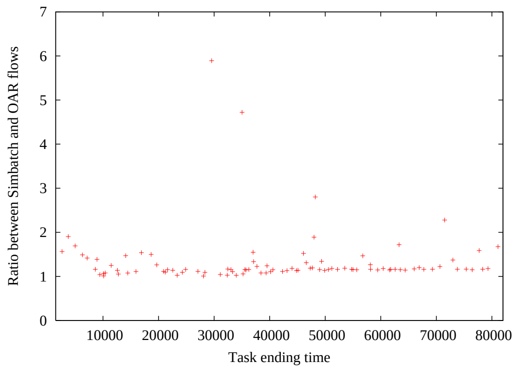


Figure 4. Flow comparison between Simbatch and OAR results.

## 3.4. 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 [1, 22, 27, 29] 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 their 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 $\mathrm{MA}_{DAG}$, manages the different workflow submissions. An overview of the new DIET architecture is shown in Figure 5.

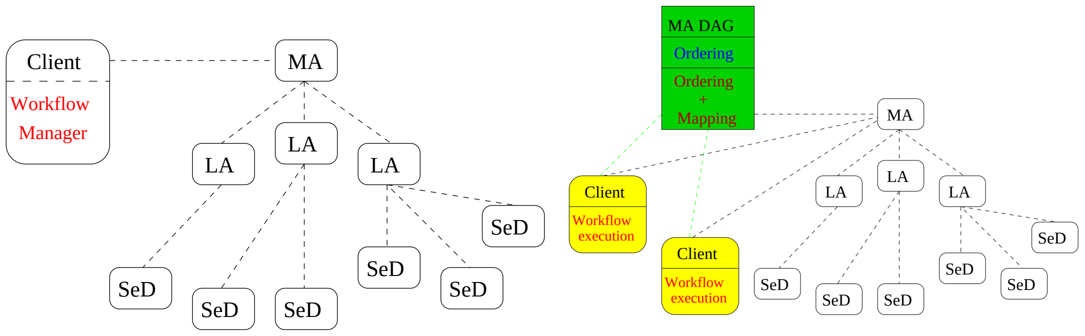


Figure 5. Software architecture of DIET workflow engine.

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.

## 4. DIET AND ADDITIONAL GRID TOOLS

DIET uses a scalable event monitoring system called LogService [17]. This monitoring service offers the capability to monitor information that must be gathered from a distributed platform. *LogComponent* attaches to a component and relays information and messages to *LogCentral*. *LogCentral* collects messages received from *LogComponents*, then it stores or sends these messages to *LogTools*. *LogTools* connect themselves to *LogCentral* and wait for messages. The main interest of LogService is that information are collected by a central point *LogCentral* that receives *LogEvents* from *LogComponents* that are attached to the component that needs to be monitored. *LogCentral* offers the service of re-sending this information to several tools (*LogTools*) which are responsible for analyzing these messages and offering a comprehensive view of the system to the user. On each component of DIET there is a *LogComponent* that sends information to *LogCentral*. VIZDIET [6] has been developed to offer visualization of

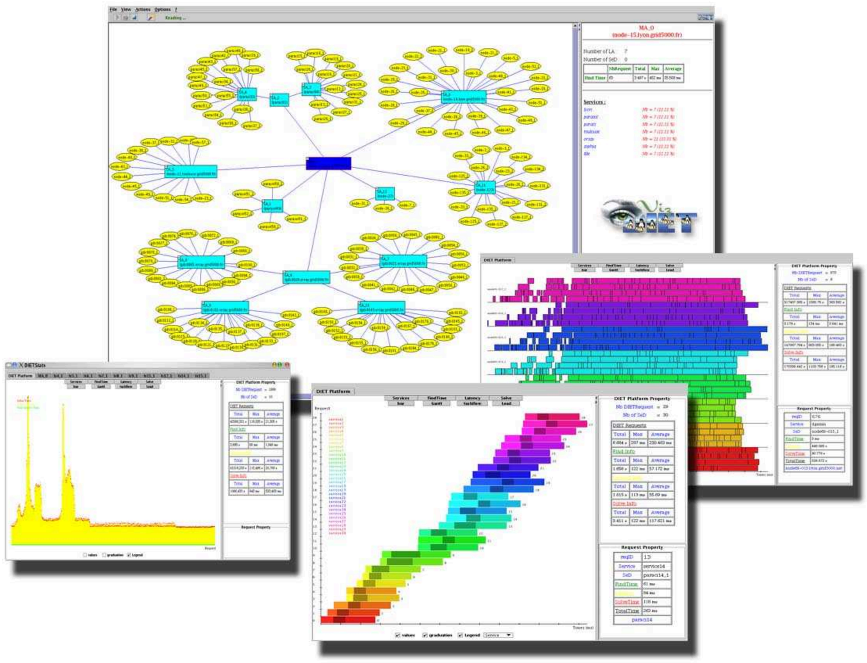


Figure 6. VIZDIET screenshots.

the DIET hierarchy. VIZDIET is linked against the *LogTools* library to be able to connect to *LogCentral* and collect all events from the DIET hierarchy.

VIZDIET gives a graphical representation of the platform as well as some quantitative and qualitative information (Figure 6) about the performance and behavior of DIET. VIZDIET is very useful as it dynamically displays the activity of the platform. Moreover it can also read a log file of a previous DIET run and replay it.

Moreover to easily manage the DIET platform, we now provide DIET Dashboard. DIET Dashboard is a set of tools written in Java that provide DIET end-user with a friendly-user interface to design, deploy and monitor the execution of applications. DIET Dashboard is an extensible set of graphical tools for the DIET community. DIET Dashboard is currently based on seven tools. (1) Workflow designer. This tool provides the user with an easy way to design and execute workflows with DIET. After being connected to a deployed DIET platform, the user can compose the different services available and link them by drag'n'drop. Once the workflow designed, the user can set its parameters and then execute this application. DIET Dashboard includes a client that can execute this workflow. The results of the workflow and the execution log are displayed. (2) Workflow log service. This tool can be used to monitor workflows execution by displaying the DAG nodes of each workflow and their states. Three states are available: waiting, running and done. The workflow log service can be used in two modes:

- local mode: can be used if the DIET grid application can access to the client's local host.
- remote mode: this mode is useful if the target platform is behind a firewall and allows only ssh connections.

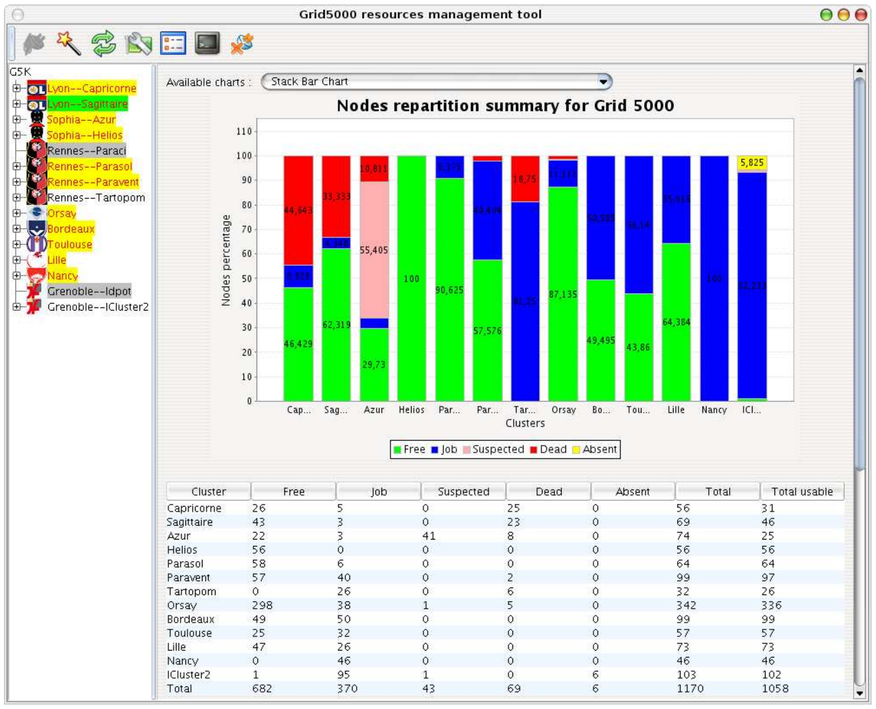
(3) DIET designer allows the user to design graphically a DIET hierarchy of schedulers and servers. Only the application characteristics are defined (agent type: Master or Local and SeD parameters). The designed application can be stored to be used with DIET mapping tool. (4) DIET mapping tool allows the user to map the allocated Grid'5000 [9] resources to a DIET application. The mapping is done in an interactive way by selecting the site then DIET agents or SeD. For each Grid'5000 site, the nodes (or hosts) are used in a homogeneous manner but the user can select a particular host if needed. (5) DIET deployment tool. This tool is a graphical interface to GoDIET [11]. It provides the basic GoDIET operations: open, launch, stop and also a monitoring mechanism to check if DIET application elements are still alive (three states are available: unknown, dead and running). As for the workflow log service, the DIET deployment tool can be used in a local or remote mode. (6) DIET resource tool. This tool was designed to manage the user grid resources which is an important aspect of grid computing. Currently this tool is used only for the Grid'5000 platform and provides several operations to facilitate the access to this platform (see Figure 7). The main features:

- Displaying the status of the platform: this feature provides information on clusters, nodes and jobs.
- Resources allocation: this feature provides an easy way to allocate resources by selecting in a Grid'5000 map the number of required nodes and the time needed for execution. The allocated resources can be stored and used with the DIET mapping tool.

(7) XMLGoDIETGenerator. This tool was designed to help the end-user creating hierarchies from existing frameworks based on the reserved resources. The user is asked to choose an experience (a framework of hierarchy) from the ones available (personal hierarchies can be added, for more information see the XMLGoDIETGenerator documentation). For each hierarchy the user has to specify the required elements involved (MA/LA/SeD). Finally a platform is generated and the user can then deploy it through the DIET deploy tool.

Some experiments have been conducted with DIET on Grid'5000 using this set of tools. Grid'5000 is a large scale experimental grid platform connecting clusters in nine different research centers in France. We have evaluated the scalability of DIET over more than one thousand processors distributed in this nation-wide grid. The experimental process used for performing these tests involved four main steps. (1) Reserve with OAR [8] 550 dual-processor nodes which will be used to run the DIET components. (2) Generate an XML file describing the reserved nodes and the desired deployment. (3) Use GoDIET [11] to deploy the hierarchy of DIET components. (4) Launch 1040 clients which continuously submit requests to the hierarchy for solving a matrix multiplication problem (DGEMM).

During this experiment, there was one local agent managing each cluster. There were a total of 540 SEDs running the same service (DGEMM), eight local agents, and one master agent.


Grid5000 resources management tool
G5K
Lyon--Capricorne
Lyon--Sagittaire
Sophia--Azur
Sophia--Helios
Rennes--Paraci
Rennes--Parasol
Rennes--Paravent
Rennes--Tartopom
Orsay
Bordeaux
Toulouse
Lille
Nancy
Grenoble--Idpot
Grenoble--ICluster2
Available charts : Stack Bar Chart
Nodes repartition summary for Grid 5000
Nodes percentage
Cap... Sag... Azur Helios Par... Par... Tar... Orsay Bo... Tou... Lille Nancy ICl...
Clusters
🟩 Free 🟦 Job ▨ Suspected 🟥 Dead 🟨 Absent

| Cluster | Free | Job | Suspected | Dead | Absent | Total | Total usable |
|---|---|---|---|---|---|---|---|
| Capricorne | 26 | 5 | 0 | 25 | 0 | 56 | 31 |
| Sagittaire | 43 | 3 | 0 | 23 | 0 | 69 | 46 |
| Azur | 22 | 3 | 41 | 8 | 0 | 74 | 25 |
| Helios | 56 | 0 | 0 | 0 | 0 | 56 | 56 |
| Parasol | 58 | 6 | 0 | 0 | 0 | 64 | 64 |
| Paravent | 57 | 40 | 0 | 2 | 0 | 99 | 97 |
| Tartopom | 0 | 26 | 0 | 6 | 0 | 32 | 26 |
| Orsay | 298 | 38 | 1 | 5 | 0 | 342 | 336 |
| Bordeaux | 49 | 50 | 0 | 0 | 0 | 99 | 99 |
| Toulouse | 25 | 32 | 0 | 0 | 0 | 57 | 57 |
| Lille | 47 | 26 | 0 | 0 | 0 | 73 | 73 |
| Nancy | 0 | 46 | 0 | 0 | 0 | 46 | 46 |
| ICluster2 | 1 | 95 | 1 | 0 | 6 | 103 | 102 |
| Total | 682 | 370 | 43 | 69 | 6 | 1170 | 1058 |

Figure 7. DIET resource tool screenshot.

13761 requests were computed by the DIET hierarchy. The scheduling heuristic was a simple round robin approach[†] that used the time since last solve for each SED to coordinate round robin behavior amongst the distributed SEDs. Figure 8 shows that the time taken by agents to schedule requests depends on the computing power and the network connection of the nodes within the cluster. There are some huge variations of response time except at *Lyon* and *Paraci*. This can be explained by the fact that the nodes used by the SEDs and agents were shared with other users. At *Lyon* and *Paraci* the nodes were reserved in an exclusive mode so that response time remains relatively constant. The main goal of this experiment was to prove that DIET can be used on large scale grids while maintaining a low response time (average of 1.9 s) despite a heavy load (1040 clients). Further experiments will be done to test and improve DIET features and performance.

[†]We do not take care of the completion time of the requests, we simply want to heavily load the platform

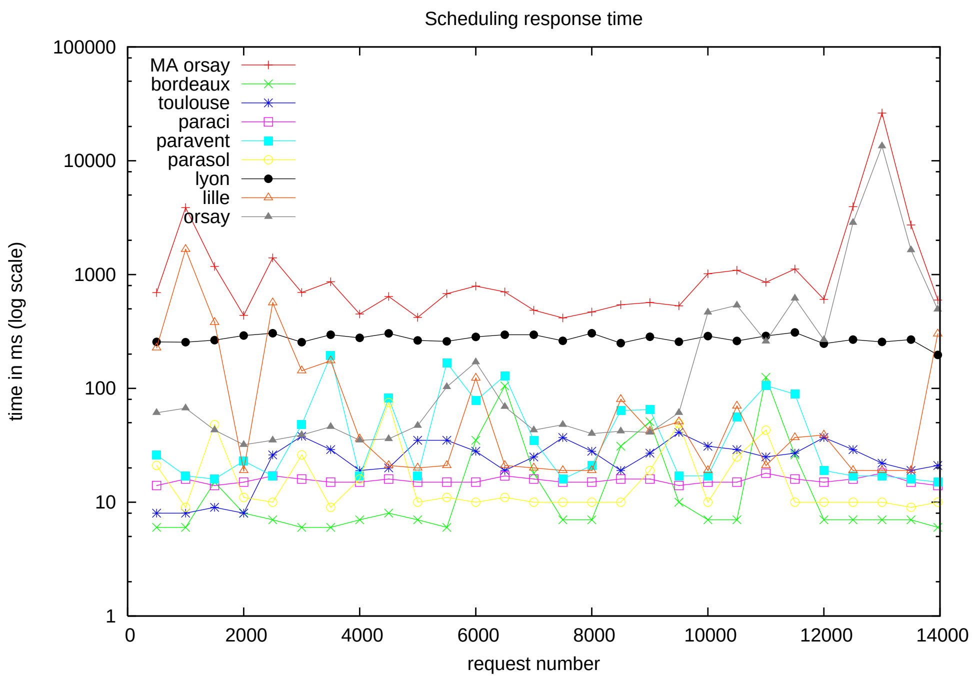


Figure 8. Response time of agents for scheduling client's requests.

## 5. DIET APPLICATIONS

Now we describe two applications using Diet. The first one is a biological application and the second one is a cosmological application.

### 5.1. A BLAST Application Using DIET

BLAST is a popular application in bioinformatics for comparing biological sequences such as nucleotides or amino-acid sequences. The aim of such comparisons is to try to determine the function of a new sequence by finding *homologies* with known sequences. A typical use of BLAST is to compare one or more sequences to one or more biological databases. Many approaches to parallelizing BLAST have been investigated [10, 5, 7, 30, 18] and three levels of parallelization can be identified. In the fine grained parallelization approach, alignment searches are performed in parallel on a single sequence pair. For the medium grained parallelization, databases are partitioned so that alignments between a sequence and each part of the database can be searched at once. With coarse grained parallelization, the input is partitioned so that multiple sequences can be compared against one or more databases at once.

Using the DIET middleware, we developed an "N-sequences versus one database" service for BLAST queries. On the client side, the multi-request files are partitioned into several smaller requests according to a user strategy (decided by choosing an existing input plug-in or a self-made one). Then, the requests are distributed over the available SEDs, which treat them independently and send back the results to the client which merges them after choosing an output plug-in. On the server side, a scheduling strategy can be applied to choose the most appropriate server to execute a request. The server then sends back the result of the execution of a BLAST implementation (including mpiBLAST when a server manages a cluster).

The next focus of the work on DIET BLAST is to introduce data replication in the database management. In the current version, we assume that every server declaring the DIET BLAST service owns the databases used for client requests. We also want to introduce a generic data partitioning information system into the DIET architecture to develop an "intelligent" request decomposition plug-in for the client. This system should be used when the input data of a problem can be divided into multi-size parts and could benefit other applications that use DIET as middleware. The last step is to implement a database partition system acting like mpiBLAST to improve the performance of the single sequence versus one large database requests.

## 5.2. Cosmological Simulation with RAMSES and GALICS

RAMSES* is a grid-based hydro solver with adaptive mesh refinement. This code is used to study large scale structure and galaxy formation: from the early universe's structure, the evolution of the position, mass, and velocity of the different particles is followed until now. The raw data produced by RAMSES are then processed using the GALICS* software (HaloMaker, TreeMaker and GalaxyMaker) to extract the halos of matter (gathering of particles), to build the evolution tree (how each particle has evolved), and finally to build galaxies.

The experiments are done on the Grid'5000 platform. RAMSES is not suited for this heterogeneous platform as it is an MPI code. It is then convenient to use the DIET middleware to provide a simpler, transparent way of using this cosmological simulator on each cluster composing the platform.

The simulation we are working on is called a zoom simulation, wherein the goal is to study in detail the evolution of the distribution of dark matter in the universe. The first part consists of using RAMSES on low resolution initial conditions (few particles) to have a global map of the different particle clusters formed from the primordial universe until now. These data are then post-processed using HaloMaker and the halos' descriptions are sent back to the user, who decides which parts may be interesting to analyze more precisely. The simulation is then rerun on all these different parts at a higher resolution (lots of particles), and on specific locations of

*http://www-dapnia.cea.fr/Phocea/Vie_des_labos/Ast/ast_sstechnique.php?id_ast=904
*http://galics.iap.fr/

the universe. The post-processing uses HaloMaker, TreeMaker and GalaxyMaker sequentially and the final results are sent back to the user for further interpretation.

The structure of this experiment is divided in three parts: the client which sends the request and analyzes the data; the servers that run the simulation; and a database containing the initial conditions. The two parts of the simulation are basically the same: they run RAMSES on initial conditions, post-process the data, and return them to the client. Therefore, many SEDs capable of managing the whole simulation may be deployed (each SED offering two services: one for each part), allowing DIET to choose the most accurate one at a given time, and bringing total transparency to the user. The user will only have to send a request through DIET, which will ask its hierarchy for the service, and run it. The access to the database will also be transparent, as only the SEDs will have to extract the initial conditions from it. Data management is one of our concerns as the amount of transferred data may be large: we may have file sizes up to 1 GB. We intend to use the JUXMEM[†] software for data management, which provides location transparency as well as data persistence in a dynamic environment. However, this part is not yet implemented in our prototype and we still use DIET for communications between the SEDs and the client, and `scp` for communications between the SEDs and the database.

## 6. CONCLUSION AND FUTURE WORK

In this paper we have presented the overall architecture of DIET, a scalable environment for the deployment on the grid of applications based on the Network Enabled Server paradigm as well as its most recent developments. Like NetSolve and Ninf, DIET provides an interface to the GridRPC API defined within the Global Grid Forum.

Our main objective is to improve the scalability of the platform using a distributed set of agents managing a large set of servers available through the network. By being able to modify the number of schedulers, we are able to ensure a level of performance adapted to the characteristics of the platform (number of clients, number and frequency of requests, performance of the target platform). The management of the platform is handled by several tools like DIET Dashboard and GODIET for the automatic deployment of the different components, LogService for monitoring, and VIZDIET for the visualization of the behavior of DIET's internals. Scheduling is of course one of the main research issue addressed within our tool. Thanks to several APIs, we are able to tune the scheduler itself to either best fit the needs of specific users or to test new heuristics for particular problems.

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 to test several DIET platforms at a large scale within the Grid'5000 project [9].

**ACKNOWLEDGEMENTS**

[†]http://juxmem.gforge.inria.fr/

Diet will not be the same middleware without the help of students, engineers, and postdoc who spent some time in our team. Related to the results presented in this paper, we would like to give special thanks to A. Bouteiller, H. Dail, P. Frauenkron, L. Huys and A. Su.

## REFERENCES

1. Amin K, von Laszewski G, Hategan M, Zaluzec NJ, Hampton S, Rossi A. GridAnt: A client-controllable grid workflow system. *hicss*, 2004; **07**:70210c.
2. Arbenz P, Gander W, Mori J. The Remote Computational System. *Parallel Computing*, 1997; **23**(10):1421–1428.
3. Arnold D, Agrawal S, Blackford S, Dongarra J, Miller M, Sagi K, Shi Z, Vadhiyar S. Users' guide to NetSolve v1.4. *Computer Science Dept. Technical Report* 2001; CS-01-467, University of Tennessee, Knoxville, TN, July 2001.
http://www.cs.utk.edu/netsolve/.
4. Arnold DC, Casanova H, Dongarra J. Innovations of the NetSolve grid computing system. *Concurrency And Computation: Practice And Experience*, 2002; **14**:1–23.
5. Bjornson RD, Sherman AH, Weston SB, Willard N, Wing J. Turboblast: A parallel implementation of blast based on the turbohub process integration architecture. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS,* TurboGenomics, Inc., 2002. 183–190.
6. Bolze R, Caron E, Desprez F, Hoesch G, Pontvieux C. A monitoring and visualization tool and its application for a network enabled server platform. In *Computational Science and Its Applications - ICCSA 2006*, Gavrilova M et al. (ed.), vol. 3984 of *LNCS* Glasgow, UK., 8-11 May 2006. Springer. 202–213.
7. Braun RC, Pedretti KT, Casavant TL, Scheetz TE, Birkett CL, Roberts CA. Parallelization of local BLAST service on workstation clusters. *FGCS*, 2001; **17**(6):745–754.
8. Capit N, Da Costa G, Georgiou Y, Huard G, Martin C, Mounié G, Neyron P, Richard O. A batch scheduler with high level components. In *Cluster computing and Grid 2005 (CCGrid05)*, 2005.
9. Cappello F, Caron E, Dayde M, Desprez F, Jeannot E, Jegou Y, Lanteri S, Leduc J, Melab N, Mornet G, Namyst R, Primet P, Richard O. 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*, Seattle, USA, November 13-14 2005; IEEE/ACM. pages 99–106.
10. Carey L, Darling AE, Feng W-chun. The design, implementation, and evaluation of mpiblast. *ClusterWorld 2003*, 2003.
11. Caron E, Kaur Chouhan P, Dail H. Godiet: A deployment tool for distributed middleware on grid'5000. In *EXPGRID workshop. Experimental Grid Testbeds for the Assessment of Large-Scale Distributed Apllications and Tools. In conjunction with HPDC-15.* IEEE (ed.), Paris, France, June 2006; 1–8
12. Caron E, Desprez F. Diet: A scalable toolbox to build network enabled servers on the grid. *International Journal of High Performance Computing Applications* 2006; **20**(3):335–352.
13. Dail H, Desprez F. Experiences with Hierarchical Request Flow Management for Network-Enabled Server Environments. *International Journal of High Performance Computing Applications* 2006; **20**(1):143–157.
14. Denis A, Perez C, Priol T. Towards high performance CORBA and MPI middlewares for grid computing. In *Proc. of the 2nd International Workshop on Grid Computing* Craig A Lee, (ed.) Springer-Verlag: Denver, Colorado, USA, November 2001; number 2242 in LNCS, 14–25.
15. DIET. Distributed Interactive Engineering Toolbox.
http://graal.ens-lyon.fr/DIET.
16. Legrand A, Marchal L, Casanova H. Scheduling distributed applications: the simgrid simulation framework. In *3rd International Symposium on Cluster Computing and the Grid*, IEEE Computer Society (ed.), May 2003; 138.
17. LogService.
http://graal.ens-lyon.fr/DIET/logservice.html.
18. Mathog DR. Parallel BLAST on split databases. Bioinformatics, September 2003; **19**(14):1865–1866.
19. Matsuoka S, Nakada H, Sato M, Sekiguchi S. Design issues of network enabled server systems for the grid. *Grid Forum, Advanced Programming Models Working Group whitepaper* 2000.
20. Nakada H, Sato M, Sekiguchi S. Design and implementations of Ninf: towards a global computing infrastructure. *Future Generation Computing Systems, Metacomputing Issue*, 1999 **15**(5-6):649–658.
http://ninf.apgrid.org/papers/papers.shtml.

21. Nakada H, Tanaka Y, Matsuoka S, Sekiguchi S. The design and implementation of a fault-tolerant RPC system: Ninf-C. In Proceeding of HPC Asia 2004, 2004; 9–18.
22. Oinn TM, Addis M, Ferris J, Marvin D, Greenwood RM, Carver T, Pocock MR, Wipat A, Li P. Taverna: a tool for the composition and enactment of bioinformatics workflow. *Bioinformatics*, November 2004; **20**(17):3045–3054.
23. Quinson M. Dynamic performance forecasting for network-enabled servers in a metacomputing environment. In *International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS'02), in conjunction with IPDPS'02*, Apr 2002.
24. Sato M, Boku T, Takahasi D. OmniRPC: a grid RPC system for parallel programming in cluster and grid environment. In Proceedings of CCGrid2003, Tokyo, May 2003; 206–213.
25. Seymour K, Lee C, Desprez F, Nakada H, Tanaka Y. The end-user and middleware APIs for GridRPC. In Workshop on Grid Application Programming Interfaces, In conjunction with GGF12, Brussels, Belgium, September 2004.
26. Shirasuna S, Nakada H, Matsuoka S, Sekiguchi S. Evaluating web services based implementations of GridRPC. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11 2002)*, July 2002. 237–245. http://matsu-www.is.titech.ac.jp/~sirasuna/research/hpdc2002/hpdc2002.p%df.
27. Singh G, Deelman E, Mehta G, Vahi K, i Su MH, Berriman GB, Good J, Jacob JC, Katz DS, Lazzarini A, Blackburn K, Koranda S. The pegasus portal: web based grid computing. In SAC '05: Proceedings of the 2005 ACM symposium on Applied computing. ACM Press:New York, NY, USA, 2005; 680–686.
28. Tanaka Y, Nakada N, Sekiguchi S, Suzumura T, Matsuoka S. Ninf-G: A reference implementation of RPC-based programming middleware for grid computing. *J. of Grid Comput.*, 2003; 1:41–51.
29. Condor Team. The directed acyclic graph manager. http://www.cs.wisc.edu/condor/dagman.
30. Wang C, Alqaralleh BA, Zhou BB, Till M, Zomaya AY. A BLAST service built on data indexed overlay network. In *e-Science*, 2005; 16–23.
31. Tanaka Y, Takemiya H, Nakada H, Sekiguchi S. Design, implementation and performance evaluation of GridRPC programming middleware for a large-scale computational grid. In *Proceedings of 5th IEEE/ACM International Workshop on Grid Computing*, 2005; 298–305.