Diet: New Developments and Recent Results*

A. Amar[1], R. Bolze[1], A. Bouteiller[1], A. Chis[1], Y. Caniou[1], E. Caron[1], P.K. Chouhan[1], G. Le Mahec[2], H. Dail[1], B. Depardon[1], F. Desprez[1], J.-S. Gay[1], A. Su[1]

[1] LIP Laboratory (UMR CNRS, ENS Lyon, INRIA, UCBL 5668) / GRAAL Project
[2] LPC / PCSV (CNRS / IN2P3 UBP Clermont-Ferrand)
Frederic.Desprez@inria.fr

Abstract. 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. The aim of this paper is to give an overview of an NES middleware developed in the GRAAL team called DIET and to describe recent developments. DIET (Distributed Interactive Engineering Toolbox) is a hierarchical set of components used for the development of applications based on computational servers on the grid.

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 GridRPC systems [31]. 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 their 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) [18] project is focused on the development of 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.

Several other Network Enabled Server (NES) systems have been developed in the past [4, 24]. Among them, NetSolve [5], Ninf [25], and OmniRPC [30] have particularly pursued research involving the GridRPC paradigm. 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

* DIET was developed with financial supports from the French Ministry of Research (RNTL GASP and ACI ASP) and the ANR (Agence Nationale de la Recherche) through the LEGO project referenced ANR-05-CIGC-11.

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 [6]. Ninf is an 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 [34, 37] or Web Services [32]. Fault tolerance is also provided using Condor and a checkpointing library [26]. The performance of the platform can be studied using a powerful tool called BRICKS. As compared to the NES systems described above, DIET is interesting because of the use distributed scheduling to provide better scalability, the ability to tune behavior using several APIs, and the use of CORBA as a core middleware.

In this paper, we present the last developments done within the DIET project that will provide the user with an efficient, scalable, and fault-tolerant system for the deployment to deploy large scale applications over the net. 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. The DIET platform deployment tool is described in Section 4 and fault-tolerance detection and recovery are explained in Section 5. The visualization of DIET's behavior on large scale platforms is described in Section 6. Finally, before a conclusion, we describe two new applications ported over DIET.

2 DIET Architecture

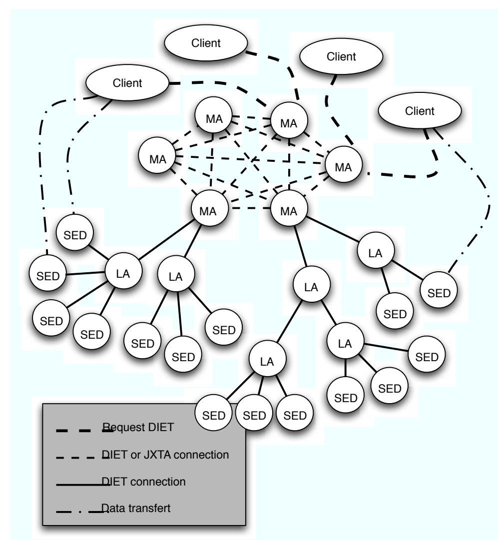


Fig. 1. DIET hierarchical organization.

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 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 [17]. Thus, CORBA has been chosen as a communication layer in DIET.

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 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 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.

`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

Table 1. Standard estimation tags used in DIET.

The standard values are to be identified based on standard estimation tags given in Table 1. 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.

The application developer 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 declared performance estimation routine. 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 sorting of the 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 has depended on a performance prediction tool called FAST [29]. In this paper, we add new functionality 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 blockage of the 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.

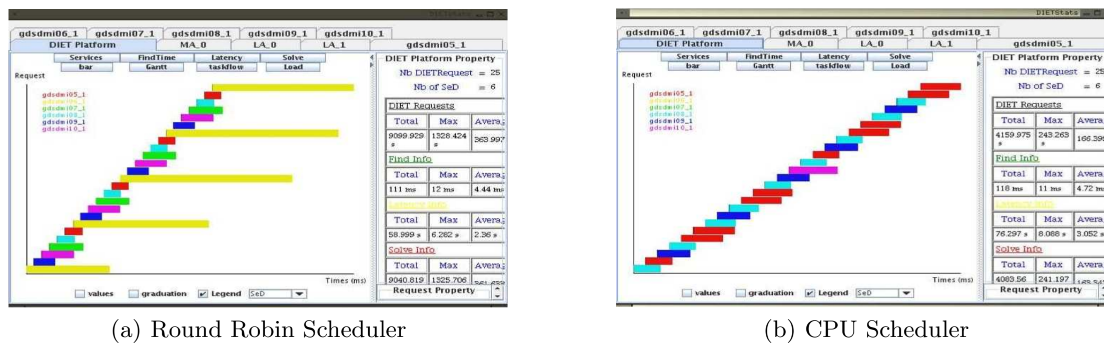
The new tool has to solve two main problems. First, it must provide basic measurements that are available regardless of the execution environment. The service developer can then rely on this collector of resource information even if no other resource services like FAST or NWS are available . Secondly, 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 logical 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 the modular design. Because the interface of the manager allows the addition of a new module in some steps, additional modules like Ganglia or NWS can easily be added.

To conclude this section and as a proof of concept, Figure 2 shows an experiment using two kinds 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 $\frac{BOGOMIPS}{1+load_average}$.



(a) Round Robin Scheduler

(b) CPU Scheduler

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

The behavior of both schedulers was also studied for requests with different inter-arrival times on an 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 the tasks are assigned on the fastest servers. The overlapping 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[3], PBS[4], or OAR[5]). 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 (walltime) needed for the reservation. In order to get the job completed as soon as possible, users can take into account the hardware (walltime), the software he can rely on (NFS for copying some data) and the actual workload on the system (number of resources to use in order to receive resources as soon as possible).

An efficient grid middleware should provide transparent access to parallel resources for the user. It must choose the best parallel resources that suit the request, tune the parallel task to 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.

Simbatch[6] is a C API which relies on the grid simulator Simgrid[20] 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.

[3] http://www-03.ibm.com/servers/eserver/clusters/software/loadleveler.html
[4] http://www.clusterresources.com/pages/products/torque-resource-manager.php
[5] http://oar.imag.fr/
[6] http://graal.ens-lyon.fr/Simbatch

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 3 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 number of processors required 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 and one can observe the error between the measured flow in a real OAR batch system and the flow obtained for the same experiment within Simbatch as a function of the submission date of the tasks.

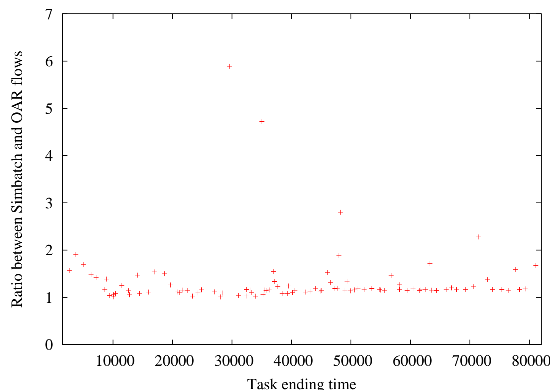


Fig. 3. Flow comparison between Simbatch and OAR results.

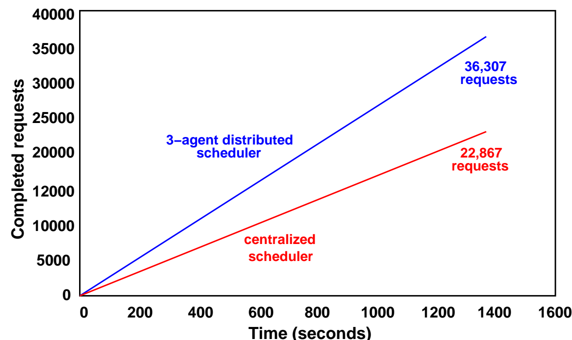


Fig. 4. Comparison of requests completed by a centralized DIET scheduler versus a three agent distributed DIET scheduler.

The high precision of Simbatch simulation results makes it possible to use it as a performance prediction module in the DIET environment. Assuming that a performance prediction function 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.

A grid environment must dialogue with batch schedulers to get information on the parallel resources in order to make performance predictions and to submit tuned parallel applications with the correct semantics.

Our work relies on the Elagi[7] library. It provides in particular the possibility to submit jobs to batch systems including Loadlever, Sun Grid Engine[8], 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[9] project (*Enabling Grids for E-science in Europe*).

The DIET batch API provides several functions. A client can explicitly ask for a parallel job, but otherwise, whenever possible DIET will choose the best available allocation to minimize a given objective function. On the server side, the resolution of the application must end with the `diet_submit_call()` which builds and submits the script to the batch scheduler.

3.4 DIET Workflow Management

A large number of scientific applications are represented by interconnected tasks which are structured based on their control and data dependencies. The workflow paradigm on grids is well adapted for representing

[7] http://grail.sdsc.edu/projects/elagi/
[8] http://www.sun.qassociates.co.uk/software-grid-engine.htm
[9] http://public.eu-egee.org/

such applications and the development of several workflow engines [3, 27, 33, 35] illustrate significant and growing interest in workflows 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-workflow.

DIET users have traditionally submitted individual tasks, but we have 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 new DIET architecture is shown in Figure 5.

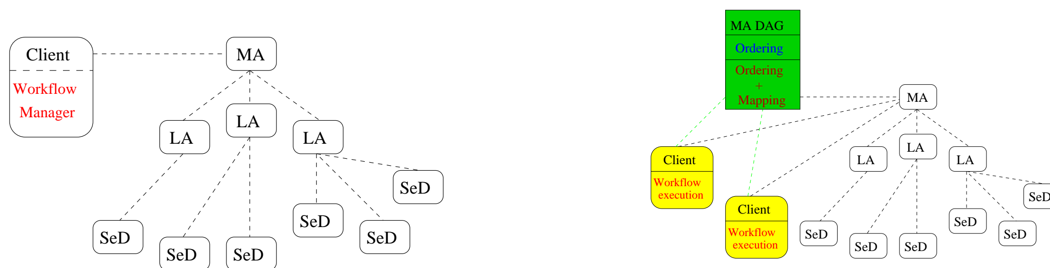


Fig. 5. Software architecture of DIET workflow engine.

The two architectures presented in the above 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 aspects. Two working modes can be used in the MA_{DAG}: in the first a complete schedule (which assign priority and mapping to each task) is provided to the client, while in the second only task priorities are returned to the client.

4 DIET Deployment

An important factor that influences the efficiency of DIET is the mapping of its components on available resources. We call mapping of the components on available resources "deployment". GoDIET [13] is designed to automate the deployment of DIET platforms and associated services for diverse grid environments. Key goals of GoDIET included portability, the ability to integrate GoDIET in a graphically-based user tool for DIET management, and the ability to communicate in CORBA with LogService [14]. GoDIET automatically generates configuration files for each DIET element while taking into account user configuration preferences and the hierarchy defined by the user, launches complimentary services (such as a name service and logging services), provides an ordered launch of components based on dependencies defined by the hierarchy, and provides remote cleanup of launched processes when the deployed platform is to be destroyed.

To show that the efficiency of an NES environment can depend on the arrangement, or deployment, of its components on available resources we performed several experiments using DIET and shown in Figure 4. These experiments were performed using 151 nodes of the Orsay cluster of the Grid'5000 testbed [10]. Depending on the number of nodes available and the number and sizes of requests, several deployments are possible. For example, we tested two deployments with 150 nodes. In the first deployment, one node is a centralized scheduler that is used to manage scheduling for the remaining 150 nodes, which are dedicated computational nodes servicing requests. In the second deployment, three nodes are dedicated to scheduling and are used to manage scheduling for the remaining 148 nodes, which are dedicated to servicing computational requests. In this test the centralized scheduler is able to complete 22,867 requests in the allotted time of about 1400 seconds, while the hierarchical scheduler is able to complete 36,307 requests in the same amount of time.

The distributed configuration performed significantly better, despite the fact that two of the computational servers are dedicated to scheduling and are not available to service computational requests. The deployment plan of components is a very important factor that influences the throughput of the environment. Thus a good planning approach is needed to arrange the resources in such a manner that when the components are deployed on the resources, the maximum number of requests can be processed in a time step. We called this process *deployment planning*. We have shown in [16] that the optimal deployment on a cluster is a **C**omplete **S**panning **d**-ary (CSD) tree; a CSD tree is a tree that is both a complete d-ary tree and a spanning tree. This result conforms with results from the scheduling literature. More importantly, we have presented an approach for determining the optimal degree d for the tree. Finding the best deployment among heterogeneous resources is a hard problem since it amounts to finding the best broadcast tree on a general graph, which is known to be NP-complete. So we presented a deployment heuristic that predicts the maximum throughput that can be achieved by the use of available nodes. The main focus of the heuristic is to construct an hierarchy so as to maximize the throughput of each node, where the throughput depends on the number of children attached as children to the node in the hierarchy. The given heuristic provides a deployment that can meet the user request demand, if user demand is at most equal to the maximum throughput.

Finally, we gave a mathematical model [12] that can analyze an existing deployment and can improve the performance of the deployment by finding and then removing the bottlenecks. This is an heuristic approach for improving deployments of NES environments in heterogeneous grid environments. The heuristic is used to improve the throughput of a deployment that has been defined by other means. The approach is iterative: in each iteration, mathematical models are used to analyze the existing deployment to identify the primary bottleneck, and the bottleneck is then removed by adding resources in the appropriate area of the hierarchy. Using this model we can evaluate a virtual deployment before making a real deployment, provide a decision builder tool (i.e., designed to compare different hierarchies or add new resources) and take into account the hierarchies' scalability.

5 DIET Fault-Tolerance

Grids are composed of many geographically distributed resources, each having its own administrative domain. These resources are gathered using a WAN or even the Internet. Those characteristics lead grids to be more error prone than other computing environments, raising the issue of fault tolerance. NetSolve and Ninf includes some fault tolerant mechanisms based on a centralized design. In this section we describe fault tolerant mechanisms incorporated in DIET that specifically target decentralized designs.

5.1 Fault Detection

Most common failure scenarios include both intermittent network failures between sites and node crashes. When considering unreliable networks, ensuring application correctness requires fault detection approaches. Failure detectors can be classified based on two criteria: time to detect a failure and accuracy. Detection

[10] http://www.grid5000.org

time represents the time between a failure and definitive suspicion by the failure detector. Accuracy is the probability of a correct answer from the failure detector when queried at a random time. Classical failure detection systems, like TCP, are based on heartbeats and timeouts. Maximum time to detect a failure depends both on the arrival date of the previous heartbeat and on the maximum delivery time. Considering a WAN or larger network, maximum delivery time is hard to bound, leading to a tradeoff between long failure detection time and poor accuracy.

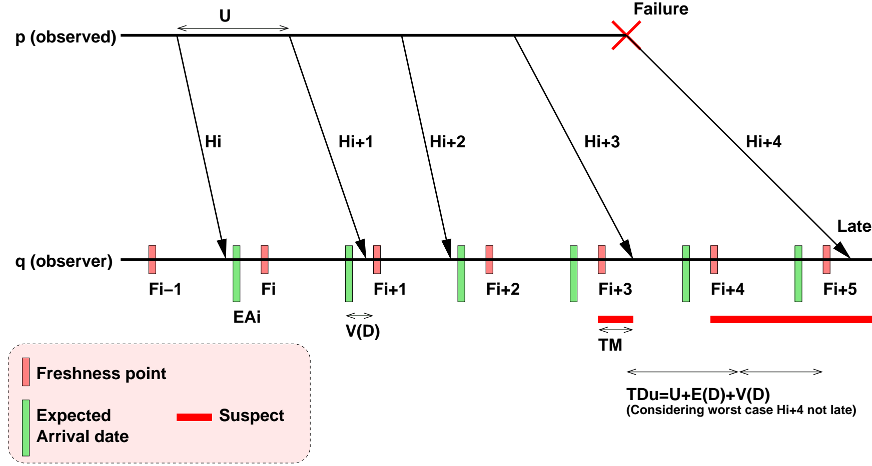


Fig. 6. Sample execution of the DIET fault detector.

DIET implements the Chandra, Toueg and Aguilera failure detector [15]. Considering a given heartbeat frequency and maximal time to detect a failure, this detector has optimal accuracy. Another benefit of this fault detector is its ability to adapt to detected network parameters and reconfigure itself according to changing network delay or message loss probability. To our knowledge, this is the first implementation of this algorithm.

Figure 6 presents a sample execution of the fault detector. The basic idea of the algorithm is as follows. Given some QoS parameters, compute a heartbeat period U to be sent by the observed process p, and some time intervals $[F_i, F_{i+1})$ on the observing process q (with respect to the local clock). At time F_i, if q has received a heartbeat $H_j, j \geqslant i$, then q trusts p for the entire time interval $[F_i, F_{i+1})$. If not, q starts suspecting p until it receives a heartbeat $H_j, j \geqslant i$. F_i is called the freshness point: any outdated message with respect to F_i is ignored. Thus maximum time to detect a failure does not depend on maximum message delay, but on average message delay E(D) (worst case when failure happens just after heartbeat emission). Parameters of the detector are the expected quality of service, namely TDu the upper bound on time to detect a failure, TMu the upper bound on average mistake duration, and $TMRl$ the lower bound on average mistake recurrence time. As freshness point F_{i+1} does not depend on the reception date of heartbeat H_i, F_{i+1} has to be set based on network parameters and expected QoS. Once a large enough sample of previous heartbeats is collected, we compute three values: EAi - the expected arrival date of Hi on q, V(D) - the variance of message delay, and Pl - the probability that a message will be lost. The variance is added to the estimated arrival date on q to set F_{i+1}. When network parameters are changing, the observing process may reconfigure the heartbeat period on p according to newly computed parameters.

DIET FD is a part of the DIET library included in every DIET entity. There is no centralized fault detector: each entity is in charge of observing its neighbors. Each observed server costs 750B of memory on the observer. As most computation is triggered by the reception of a heartbeat, and the typical heartbeat period is less than one heartbeat per 5s, the computational cost per observed service is marginal. Heartbeats are very small UDP messages (40 bytes including UDP headers), thus the impact on the network of the fault

detection service is small. DIET FD is connected to DIET LogService and VizDIET and may be used to collect statistics on network parameters and grid reliability.

5.2 MA Topology Recovery

Compared to other GridRPC systems, DIET uses a decentralized hierarchical infrastructure. Master Agents and Local Agents are organized using a tree topology. Each Agent is responsible for monitoring its neighbors and reports failures to the GoDIET component. The consequence of the failure of an agent is a disconnection of the tree topology: some available services are not found by client service requests. In order to reconnect the tree, each agent keeps the list of its f ancestors. When detecting failure of its parent, the agent tries to reconnect to the nearest (in the tree) alive ancestor. If this ancestor has also failed, it then tries with the second nearest until it is able to reconnect the tree. Thus, the algorithm is able to recover without central coordination from $f-1$ simultaneous failures. During recovery, some available services may not be found. This is the same property as in auto-stabilization (except that we are considering only the crash of nodes and message loss). When reconnected, the agent updates its ancestor list from its new parent. When the root of a tree has failed, GoDIET is in charge of replacing the MA and notifies any client to use the new MA instead of the failed one.

5.3 Checkpoint/restart Mechanism

Unexpectedly, the most intrusive failures are not those hitting infrastructure such as the MA and LAs but the computing nodes [19]. This is mainly due to the large amount of lost computation time. Process replication and process checkpointing are two well known techniques to decrease the amount of lost computation in case of crash failure. In replication, the same program is running on several hosts. Any input of the program is atomically broadcasted to all the replicas. When a failure hits the main process, one of the replicas is promoted as the main process. Whenever a failure occurs, a new replica is created to replace the missing one. Thus, having f replicas is sufficient to tolerate $f-1$ simultaneous failures, but divides the available computing power by f. Checkpoint based fault tolerance relies on taking periodic snapshot of the state of a process, saving it to another (safe) place. In case of a failure, the process state is recovered from the checkpoint. This is the approach used in NetSolve and Ninf [2, 26].

In Ninf the Condor checkpoint library is used. Checkpoint data are stored on a stable checkpoint server holding all recovery data. If a failure hits the checkpoint server, it is no longer possible to recover from any other failure. In order to solve this issue the checkpoint data have to be replicated. [28] suggests the use of computing nodes to host replicated checkpoint data. In DIET, checkpoint data are considered equivalent to persistent data and distributed on computing nodes using JuxMem. JuxMem manages data persistence across failures by replicating it on several computing nodes.

Another important issue in checkpointing is software development cost. On the one hand automatic checkpointing has low software development cost but high overhead. On the other hand application checkpointing is tightly adapted to the algorithm but requires new development for any new algorithm. The DIET checkpoint API is designed to allow both simplicity and performance. From the client point of view, checkpointing and fault tolerance are fully automatic. The fault tolerance management is silently included in the usual RPC mapping layer of the DIET client library. On the service side, each computational service can choose whether it provides its own checkpoint mechanism to DIET (it only has to register checkpoint files and notify the SeD when a checkpoint is ready to be stored), or it can rely on DIET automatic checkpointing. In this case, the service is linked with the Condor Stand-alone Checkpoint Library [21] to periodically create a checkpoint file that can be restarted on any compatible architecture.

6 DIET Visualization and Large Scale Validation

DIET uses an event monitoring system called LogService [22]. 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 need 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 send information to *LogCentral*. VizDIET [8] has been developed to offer visualization of the DIET hierarchy. VizDIET implements the *LogTools* library to be able to connect to *LogCentral* and collect all events from the DIET hierarchy (Figure 7). VizDIET gives a graphical representation of the platform as well as some quantitative and qualitative information (Figure 8) about the performance and behavior of DIET. VizDIET is very useful to understanding the behavior and performance of DIET 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.

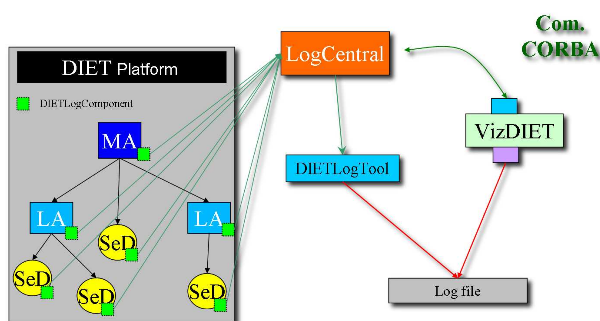


Fig. 7. The LogService mechanism in DIET.

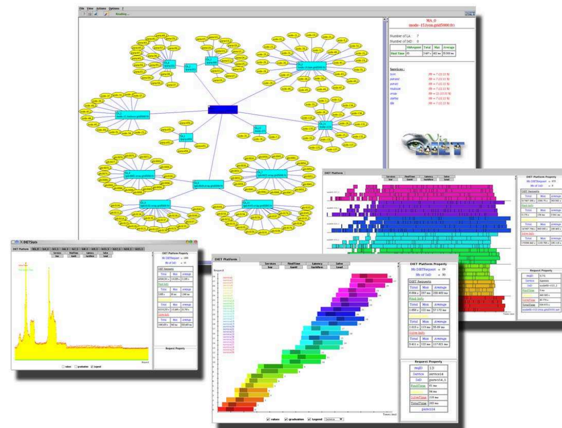


Fig. 8. VizDIET screenshots.

Recently some tests have been done with DIET on Grid'5000 [11]. We have evaluated the scalability of DIET over more than one thousand processors distributed in a nation-wide grid. The experimental process used for performing these tests involved four main steps. (1) Reserve with OAR [10] 550 dual-processor nodes that will be used to run the DIET components. (2) Generate an XML file describing the reserved nodes and the desired deployment. (3) Use GoDIET [13] 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. 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 9 shows that the time taken by agents to schedule requests depends on the computing power of the nodes on 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.

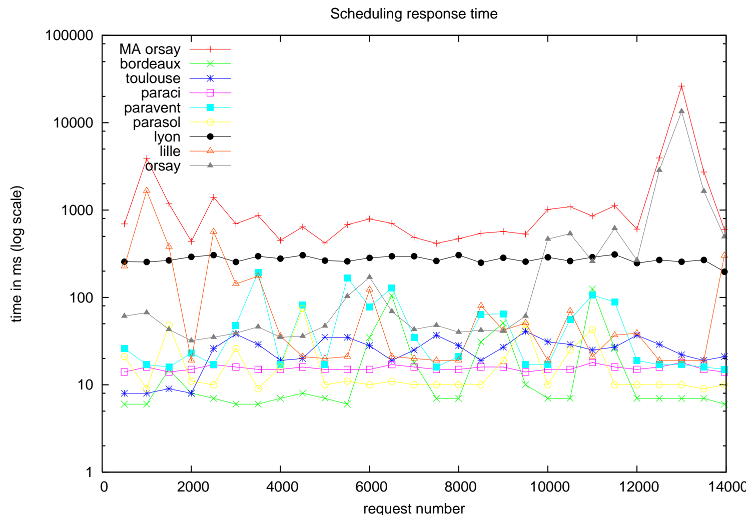


Fig. 9. Response time of agents for scheduling client's requests.

7 DIET Applications

7.1 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 [9, 1, 7, 36, 23] 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 SeD's, 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.

7.2 Cosmological Simulation with RAMSES and Galics

RAMSES[11] 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[12] 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 suited for this 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.

The simulation we're 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, 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 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 who 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 chose 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[13] 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 SeD and the database.

8 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 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.

[11] http://www-dapnia.cea.fr/Phocea/Vie_des_labos/Ast/ast_sstechnique.php?id_ast=904
[12] http://galics.iap.fr/
[13] http://juxmem.gforge.inria.fr/

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 [11].

References

1. L. Carey A. E. Darling and W. chun Feng. The design, implementation, and evaluation of mpiblast. In *Cluster-World 2003*, 2003.
2. A. Agbaria and J.S. Plank. Design, implementation, and performance of checkpointing in netsolve. *dsn*, 00:49, 2000.
3. K. Amin, G. von Laszewski, M. Hategan, N.J. Zaluzec, S. Hampton, and A. Rossi. GridAnt: A Client-Controllable Grid Workflow System. *hicss*, 07:70210c, 2004.
4. P. Arbenz, W. Gander, and J. Mori. The Remote Computational System. *Parallel Computing*, 23(10):1421–1428, 1997.
5. D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4. Computer Science Dept. Technical Report CS-01-467, University of Tennessee, Knoxville, TN, July 2001. http://www.cs.utk.edu/netsolve/.
6. D.C. Arnold, H. Casanova, and J. Dongarra. Innovations of the NetSolve Grid Computing System. *Concurrency And Computation: Practice And Experience*, 14:1–23, 2002.
7. R.D. Bjornson, A.H. Sherman, S.B. Weston, N. Willard, and J. Wing. Turboblast: A parallel implementation of blast based on the turbohub process integration architecture. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS*, pages 183–190. TurboGenomics, Inc., 2002.
8. R. Bolze, E. Caron, F. Desprez, G. Hoesch, and C. Pontvieux. A monitoring and visualization tool and its application for a network enabled server platform. In M. Gavrilova et al., editor, *Computational Science and Its Applications - ICCSA 2006*, volume 3984 of *LNCS*, pages 202–213, Glasgow, UK., 8-11 May 2006. Springer.
9. R. C. Braun, K. T. Pedretti, T. L. Casavant, T. E. Scheetz, C. L. Birkett, and C. A. Roberts. Parallelization of local BLAST service on workstation clusters. *FGCS*, 17(6):745–754, 2001.
10. N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mouni, P. Neyron, and O. Richard. A batch scheduler with high level components. In *Cluster computing and Grid 2005 (CCGrid05)*, 2005.
11. 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 99–106, Seattle, USA, November 13-14 2005. IEEE/ACM.
12. E. Caron, P. K. Chouhan, and A. Legrand. Automatic Deployment for Hierarchical Network Enabled Server. In *The 13th Heterogeneous Computing Workshop (HCW 2004)*, Santa Fe. New Mexico, April 2004.
13. E. Caron, P. Kaur Chouhan, and H. Dail. Godiet: A deployment tool for distributed middleware on grid'5000. In IEEE, editor, *EXPGRID workshop. Experimental Grid Testbeds for the Assessment of Large-Scale Distributed Apllications and Tools. In conjunction with HPDC-15.*, pages 1–8, Paris, France, June 19th 2006.
14. Eddy Caron and Frédéric Desprez. Diet: A scalable toolbox to build network enabled servers on the grid. *International Journal of High Performance Computing Applications*, 20(3):335–352, 2006.
15. W. Chen, S. Toueg, and M. Kawazoe Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computing*, 51(1):13–32, 2002.
16. P. K. Chouhan, H. Dail, E. Caron, and F. Vivien. Automatic Middleware Deployment Planning on Clusters. *International Journal of High Performance Computing Applications*, 2007. To appear.
17. A. Denis, C. Perez, and T. Priol. Towards high performance CORBA and MPI middlewares for grid computing. In Craig A. Lee, editor, *Proc. of the 2nd International Workshop on Grid Computing*, number 2242 in LNCS, pages 14–25, Denver, Colorado, USA, November 2001. Springer-Verlag.
18. DIET. Distributed Interactive Engineering Toolbox. http://graal.ens-lyon.fr/DIET.
19. 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 *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 39, Washington, DC, USA, 2004. IEEE Computer Society.
20. A. Legrand, L. Marchal, and H. Casanova. Scheduling distributed applications: the simgrid simulation framework. In IEEE Computer Society, editor, *3rd International Symposium on Cluster Computing and the Grid*, page 138. IEEE Computer Society, May 2003.
21. M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the condor distributed processing system. Technical Report 1346, University of Wisconsin-Madison, 1997.

22. LogService. http://graal.ens-lyon.fr/DIET/logservice.html.
23. D.R. Mathog. Parallel blast on split databases. *Bioinformatics*, 19(14):1865–1866, September 2003.
24. S. Matsuoka, H. Nakada, M. Sato, , and S. Sekiguchi. Design Issues of Network Enabled Server Systems for the Grid, 2000. Grid Forum, Advanced Programming Models Working Group whitepaper.
25. 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. http://ninf.apgrid.org/papers/papers.shtml.
26. H. Nakada, Y. Tanaka, S. Matsuoka, and S. Sekiguchi. The Design and Implementation of a Fault-Tolerant RPC System: Ninf-C. In *Proceeding of HPC Asia 2004*, pages 9–18, 2004.
27. 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 workflow. *Bioinformatics*, 20(17):3045–3054, nov 2004.
28. J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–980, 1998.
29. M. Quinson. 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.
30. M. Sato, T. Boku, and D. Takahasi. OmniRPC: a Grid RPC System for Parallel Programming in Cluster and Grid Environment. In *Proceedings of CCGrid2003*, pages 206–213, Tokyo, May 2003.
31. 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.
32. S. Shirasuna, H. Nakada, S. Matsuoka, and S. Sekiguchi. Evaluating Web Services Based Implementations of GridRPC. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11 2002)*, pages 237–245, July 2002. http://matsu-www.is.titech.ac.jp/~sirasuna/research/hpdc2002/hpdc2002.pdf.
33. 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.
34. Y. Tanaka, N. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *J. of Grid Comput.*, 1:41–51, 2003.
35. Condor Team. The directed acyclic graph manager. http://www.cs.wisc.edu/condor/dagman.
36. C. Wang, B.A. Alqaralleh, B.B. Zhou, M. Till, and A.Y. Zomaya. A blast service built on data indexed overlay network. In *e-Science*, pages 16–23, 2005.
37. Y. Tanaka and H. Takemiya and H. Nakada and S. Sekiguchi. 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*, pages 298–305, 2005.