

International Journal of High Performance Computing Applications

<http://hpc.sagepub.com>

Diet: A Scalable Toolbox to Build Network Enabled Servers on the Grid

E. Caron and F. Desprez


International Journal of High Performance Computing Applications 2006; 20; 335

DOI: 10.1177/1094342006067472

The online version of this article can be found at:

<http://hpc.sagepub.com/cgi/content/abstract/20/3/335>

Published by:

 SAGE Publications

<http://www.sagepublications.com>

Additional services and information for *International Journal of High Performance Computing Applications* can be found at:

Email Alerts: <http://hpc.sagepub.com/cgi/alerts>

Subscriptions: <http://hpc.sagepub.com/subscriptions>

Reprints: <http://www.sagepub.com/journalsReprints.nav>

Permissions: <http://www.sagepub.com/journalsPermissions.nav>

DIET: A SCALABLE TOOLBOX TO BUILD NETWORK ENABLED SERVERS ON THE GRID

E. Caron
F. Desprez

LIP LABORATORY/GRAAL PROJECT, UMR CNRS,
ENS LYON, INRIA, UNIV. CLAUDE BERNARD, LYON 1,
FRANCE
(FREDERIC.DESPRES@ENS-LYON.FR)

Abstract

Among existing grid middleware approaches, one simple, powerful, and flexible approach consists of using servers available in different administrative domains through the classical client-server or Remote Procedure Call (RPC) paradigm. Network Enabled Servers 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 a middleware developed by the GRAAL team called DIET (for Distributed Interactive Engineering Toolbox). DIET is a hierarchical set of components used for the development of applications based on computational servers on the grid.

Key words: Grid enabled server systems, scalable scheduling, resource management

1 Introduction

Large computational problems arising from numerical simulation or life sciences can now be solved through the Internet using grid middleware (Berman, Fox, and Hey 2003; Foster and Kesselman 2004). Several approaches co-exist to port applications on grid platforms such as classical message-passing (MPICH-G; Keller et al. 2003), batch processing (Sun Microsystems; Thain, Tannenbaum, and Livny 2004) and web portals (Kapadia, Robertson, and Fortes 1998; Good and Goux 2000; Haupt, Akarsu, and Fox 2000).

Among existing middleware approaches, one simple, powerful, and flexible approach consists of using servers available in different administrative domains through the classical client-server or Remote Procedure Call (RPC) paradigm. Network Enabled Servers (NES) (Nakada, Sato, and Seiguchi 1999; Matsuoka and Casanova 2000; Arnold et al. 2001) implement this model also called GridRPC (Seymour et al. 2004). Clients submit computation requests to a scheduler whose goal is to find a server available on the grid. Scheduling is frequently applied to balance the work load 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. As a result of the growth of network bandwidth and the reduction of network latency, small computation requests can now be sent to servers available on the grid. To make effective use of today's scalable resource platforms, it is important to ensure scalability in the middleware layers as well.

The Distributed Interactive Engineering Toolbox (DIET) project (Caron et al. 2002; <http://graal.ens-lyon.fr/DIET>) is focused on the development of scalable middleware by 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 (problem to be solved, size of the data involved), the performance of the target platform (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 dependences exist between different requests.

This paper is organized as follows. After an introduction to the GridRPC programming paradigm used on this software platform, we present the overall architecture of DIET and its main components. We give detail about their connection which can be either static or dynamic. Then, in Sec-

tion 4 we present the architecture of our performance evaluation and prediction system based on the Network Weather Service (NWS). The way DIET schedules the clients' request is discussed in Section 5. Then we discuss the issue of data management by presenting two approaches (hierarchical and peer-to-peer). In Section 7, we present some tools used for the deployment and monitoring of the platform. Finally, and before a conclusion and our future work, we present the related work on Network Enabled Servers.

2 GridRPC Programming Model

The GridRPC approach (Seymour et al. 2004) is a good candidate to build Problem Solving Environments on a computational grid. It defines an API and a model to perform remote computation on servers. In such a paradigm, a client can submit problems to an agent that chooses the best server amongst a set of candidates, given information about the performance of the platform gathered by an information service. The choice is made using static and dynamic information about software and hardware resources. Requests can then be processed by sequential or parallel servers. This paradigm is close to the RPC model.

The GridRPC API is a grid form of the classical Unix RPC approach. It has been designed by a team of researchers within the Global Grid Forum. It instructs the client API to send a request to a Network Enabled Server implementation. Requests are sent through synchronous or asynchronous calls. Asynchronous calls allow a non-blocking execution and thus provide a level of parallelism between servers. A function handle represents a binding between a problem name and an instance of such a function available on a given server. Of course several servers can provide the same function (or service) and load balancing can be done at the agent level before the binding. Then session IDs can be manipulated to get information about the status of previous non-blocking requests. Wait functions are also provided for a client to wait for a specific request to complete. This API is instantiated by several middlewares such as DIET, Ninf, NetSolve, and XtremWeb.

3 DIET Architecture

3.1 DIET Aim and Design Choices

The aim of our project is to provide a toolbox that will allow different applications to be ported efficiently over the Grid and to allow our research team to validate theoretical results on scheduling or on high performance data management for heterogeneous platforms. Thus our design follows the following principles:

Scalability When the number of requests grows, the agent becomes the bottleneck of the platform. The machine hosting this important component has to be powerful enough but the distribution of the scheduling component is often a better solution. There is of course a trade-off that needs to be found for the number (and location) of schedulers depending on various parameters such as number of clients, frequency of requests, number of servers, performance of the target platform, etc.

Simple improvement The goal of a toolbox is to provide a software environment that can be easily adapted to match users' needs. Several API have to be carefully designed at the client level, at the server level, and sometimes even at the scheduler level. These API will be used by expert developers to either plug a new application into the grid or to improve the tool for an existing application.

Ease of development The development of such a large environment needs to be done using existing middleware that will ease the design and that will offer good performance at a large scale. We chose to use CORBA as a main low level middleware (OmniORB), LDAP, and several open-source software suites like NWS, OAR, ELAGI, SimGrid, etc.

3.2 Hierarchical Architecture

The DIET architecture is based on a hierarchical approach to provide scalability. The architecture is flexible 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++. An **SeD**, or server daemon, provides the interface to computational servers and can offer any number of application specific computational services. An 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)**, several **agents (A)**, and **local agents (LA)**. Figure 1 shows an example of a DIET hierarchy.

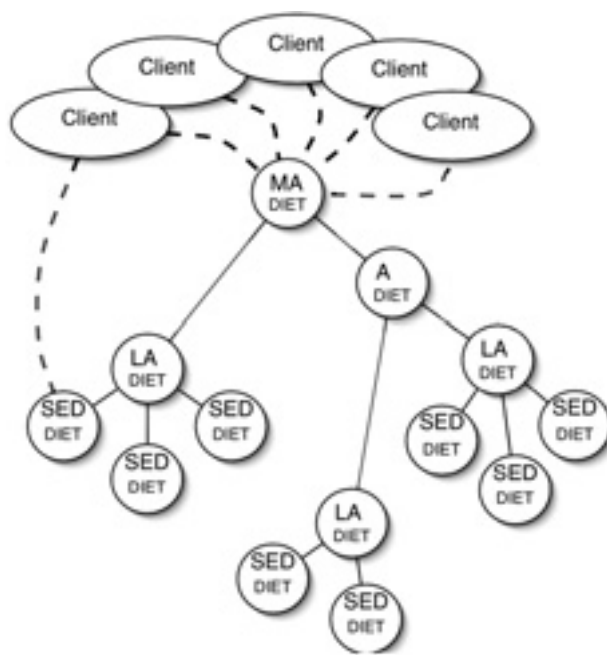


Fig. 1 DIET hierarchical organization.

A **master agent** is the 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. The 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 datasets specifically needed by the application. The 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 (computation cost, communication cost, machine load, etc.). 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 5.

Finally, NES environments like Ninf and NetSolve use a classic socket communication layer. Nevertheless, several problems to this approach have been pointed out such

as the lack of portability or the limitation of opened sockets. A distributed object environment, such as CORBA (Henning and Vinoski 1999) has been proven to be a good base for building applications that manage access to distributed services. It provides transparent communications 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, communication layers being well optimized in most CORBA implementations (Denis, Perez, and Priol 2001). Thus, CORBA has been chosen as a communication layer in DIET.

3.3 DIET Peer-To-Peer Extension

The aim of DIET_j is to dynamically connect together distributed DIET hierarchies at a large scale. This new architecture has the following properties:

Connecting hierarchies dynamically for scalability To increase the scalability of DIET over the grid, we now dynamically build a multi-hierarchy by connecting the entry points of the hierarchies (master agents), and thus research institutes and cities, together. Note that the multi-hierarchy is built on-demand by a master agent only if it fails to retrieve a service requested by a client inside its own hierarchy. Moreover, a client can now dynamically discover one or several master agents when looking for a service, and thus connect the server with the best latency and locality.

Sharing the load on the master agents So, the entry point for each client is now dynamically chosen, thus better sharing the load through master agents. Master agents are connected in an unstructured peer-to-peer fashion (without any mechanism of maintenance, routing, or group membership).

Gathering services on a large scale Services are now gathered on-demand, thus providing clients with an entry point to resources of aggregated hierarchies in a transparent way.

The DIET_j architecture, shown in Figure 2, is divided into two parts. The JXTA part includes the MA_j, the SeD_j, and the client_j; all these elements are peers on the JXTA virtual network and communicate together through it. The interface part is where Java (JXTA native language) and C++ (DIET native language) must cooperate. The technology used is JNI, which allows a Java program to call functions written in C++. A quick description of main JXTA features is available in our research report (Caron



Fig. 2 DIET_J architecture.

et al. 2005). Now, we introduce the different elements built on top of JXTA and their behavior.

Based on results presented by Jan and Noblet (2004) we believe JXTA pipes offer the right trade-off between transparency and performance for our architecture. Our implementation is based on JXTA 2.3 which minimizes the latency of the JXTA pipes, according to Jan and Noblet (2004).

The client_j. The client_j is a JXTA peer. When looking for a given service, it discovers one or several MA_j by their JXTA advertisement, chooses and binds one of them and sends it a request (encapsulated in a JXTA message) for this service. It waits for the MA_j's response. Once the response is received, it extracts the reference of the SeD(s)_j found by this MA_j. It binds one available SeD_j and sends to it the problem (encapsulated in a JXTA Message) to be solved by the SeD_{DIET}. Finally, the client_j extracts the result of the computation from the response.

The SeD_j. The SeD_j is a JXTA peer that allows the clients_j to send computation requests including data needed for the computation to the SeD_{DIET}, allowing in addition passage through firewalls, if any, between the client and the SeD. The SeD_j loads the SeD_{DIET}, and waits for client_j requests. When a JXTA message is received, the SeD_j extracts the problem and the data and calls the SeD_{DIET} to solve the problem. The result returned by the SeD_{DIET} is encapsulated in a JXTA message and sent back to the client_j.

The multi-MA and the MA_s. One multi-MA is composed of all MA_s running at a given time over the network and reachable from a first MA_j. The MA_j is able to dynamically connect with these other MA_s. Each MA_j is known on the JXTA network by an advertisement with a name common to all of them (DIET_MA) that is published at the beginning of its life. This advertisement is published with a short lifetime to avoid

clients_{*j*} (or other MAs_{*j*}) to try to bind an already stopped MA_{*j*}, and thus easily take into account the dynamicity of the platform.

The MA_{*j*} loads the MA_{DIET}, periodically re-publishes its advertisement, waiting for requests. When receiving a client_{*j*}'s request, it submits the problem description to its MA_{DIET}. If the submission to the DIET hierarchy retrieves no SeD with this service, the MA_{*j*} builds a multi-hierarchy by discovering others MA_{*j*} and propagates the request to them. When the MA_{*j*} has received responses from all other MAs_{*j*}, the responses are encapsulated in a JXTA message and sent back to the client_{*j*}.

Dynamic connections are only used between the client and the master agents, between the client and the SeD, and between the master agents themselves (using JXTA pipes advertisements). The communication between the agents inside one hierarchy are still static as we believe that small hierarchies are installed within each administrative domain. At the local level, performances are not so fluctuant and new elements are not so frequently added.

4 Performance Evaluation

Scheduling tasks on computers comes down to mapping task requirements to system availability. We now describe these more precisely. Requirements of routines group principally by the time and the memory space necessary for their execution, as well as the amount of generated communication. These requirements depend naturally on the chosen implementation and on input parameters of the routine, but also on the machine on which the execution takes place. System availability information captures the number of the machines and their speed, as well as their status (down, available, or allocated through a batch system). One must also know the topology, the capacity, and the protocols of the network connecting these machines. From the scheduling point of view, the actual availability and performance of these resources is more important than their previous use or the theoretical peak performance.

The goal of FAST (Quinson 2002) is to constitute a simple and consistent Software Development Kit (SDK) for providing client applications with accurate information about task requirements and system performance, regardless of how these values are obtained. The library is optimized to reduce its response time, and to allow its use in an interactive environment. FAST is not intended to be a scheduler by itself and provides no scheduling algorithm or facility. It only tries to provide an external scheduler with all information needed to make accurate and dynamic scheduling decisions.

Figure 3 gives an overview of FAST's architecture, which is composed of two main parts. On the bottom of

the figure, a benchmarking program is used to discover the routine's requirements on every machine in the system. Then, on top, a shared library provides accurate forecasting to the client application. This library is divided into two submodules: the right one on the figure forecasts the system performance capabilities while the left one uses the routine's requirements models. Figure 3 shows that FAST uses principally two types of external tools (in gray): A system monitoring tool, the Network Weather Service (NWS; Wolski, Spring, and Hayes 1999), and a distributed database. The first one is used to get the system performance capabilities while the second is used to store data computed at the installation phase about routine's needs. Both types of tools are fully pluggable, and adding support for a new distributed database system or a new monitoring tool is very simple.

The NWS (Wolski, Spring, and Hayes 1999) is a project led by Prof. Wolski at the University of California, Santa-Barbara. It constitutes a distributed set of sensors and statistical forecasters that capture the current state of each platform, and predict its future behavior. It is possible to monitor the latency and throughput of any TCP/IP link, the CPU load, the available memory or the disk space on any host. Concerning the CPU load, the NWS can not only report the current load, but also the time-slice a new process would acquire at startup. In order to benefit from a solid and well tested basis, the main monitoring system used is the NWS. However, for the sake of completeness, the monitoring acquisition mechanism is easily pluggable, allowing FAST to obtain information from other sources. For example, to ease the installation of FAST, it is possible to get information about the CPU load from a limited internal sensor when the NWS is not available for a given platform. In its current version, FAST can monitor the CPU and memory load of hosts, as well as the latency and bandwidth of any TCP link. In addition to the NWS, it can also report the number of CPUs on each host to ease the comparison. Monitoring new resources such as free disk space or non-TCP links should be relatively easy in the FAST framework.

At FAST install time, a list of problems of interest are specified along with their interfaces; FAST then automatically performs a series of macro-benchmarks which are stored in a database for use in the DIET scheduling process. For some applications, a suite of automatic macro-benchmarks cannot adequately capture application performance. In these cases, DIET also allows the server developer to specify an application-specific performance model to be used by the SeD during scheduling to predict performance. Although the primary targeted application class consists of sequential tasks, this approach has been successfully extended to address parallel routines as well, as explained in more details (Desprez, Quinson, and Suter 2001).

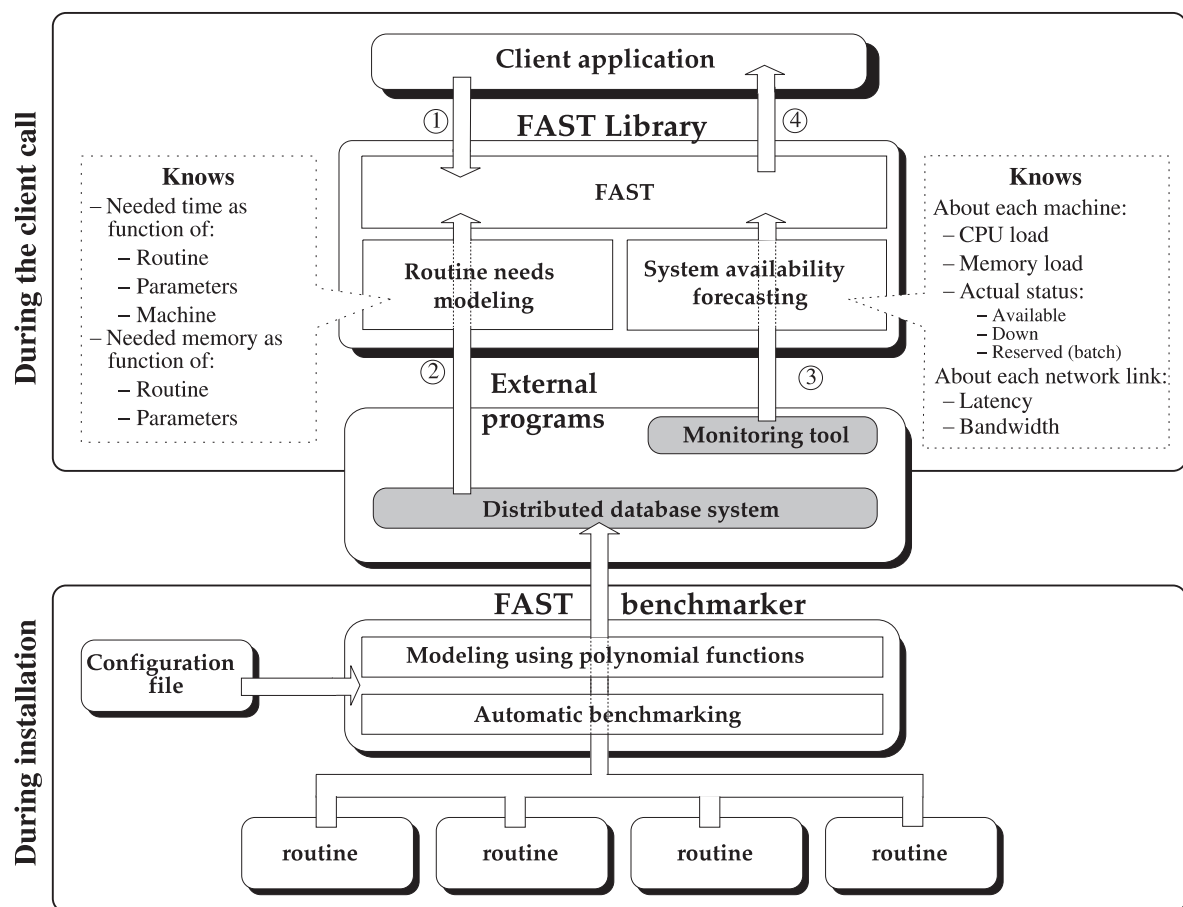


Fig. 3 FAST's architecture.

5 Scheduling

Scheduling is one of the most important issues to be solved in such an environment. Classical NES algorithms use First Come First Served approaches with the goal of minimizing the turnaround time of requests or the make-span of one application. The distributed approach chosen in the DIET platform allows the study of other algorithms where some intelligence can be put at various levels of the hierarchy.

5.1 DIET Distributed Scheduling

The primary interest of the DIET scheduling approach lies in its distribution, both in terms of collaborative decision

making and in terms of distribution of information important to the scheduling decision. We return to the general process of servicing a request to provide greater details. When the MA receives a client request, it: (1) verifies that the service requested exists in the hierarchy; (2) collects a list of its children that are thought to offer the service; and (3) forwards the request on those subtrees. Local agents use the same approach for forwarding the request to their children, whether the children are other agents or SeDs. Agents obtain information on services available in subtrees during the deployment process. When a SeD or agent starts up, it joins the DIET hierarchy by contacting its parent agent (located by a string-based name in a naming service). The parent adds the new child to its list of children and records which services are available via that child. The

parent need not track whether the service is provided directly by the child (if the child is a server) or by another server in the child's subtree (if the child is an agent); it suffices to know that the service is available *via* the child. Thus if an agent has N children and the DIET hierarchy offers a total of M services, the most hierarchy information any agent in the tree will store is $N \times M$ service/child mappings.

When an agent forwards a request to its children, it sets a timer restricting the amount of time to wait for child responses. This avoids a deadlock in the hierarchy based on one failed or slow-to-respond server. Eventually, a child will be forgotten if it is unresponsive for long enough.

SeDs are responsible for collecting and storing all of their own performance and status data. Specifically, the SeD stores a list of problems that can be solved on it, a list of any persistent data that are available locally to the server, and status information such as the number of requests currently running on the SeD and the amount of time elapsed since the last request. When a request arrives at an SeD, the SeD creates a response object containing both status information and performance data. SeDs are capable of collecting dynamic system availability metrics from the NWS (Wolski, Spring, and Hayes 1999) or can provide application-specific performance predictions using the performance evaluation module FAST (Quinson 2002) (presented in the previous section).

After the SeDs have formulated a response to the request, they send their response to their parent agent. Each agent is responsible for aggregating the responses of its children and forwarding on a sorted list of responses to the next level in the hierarchy. DIET normally returns to the user multiple server choices, sorted in order of the predicted desirability of the servers. The number N of servers to return to the client is configurable, but is of course limited by the total number of servers managed by the DIET hierarchy. Since agents have no global knowledge of the DIET hierarchy, to ensure a full list can be returned to the client each agent must return a sorted list of its N best child responses (or less if the agent subtree contains less than N servers).

The agent sorting process uses an efficient binary tree with each child node placed as the leaves. In the case of a server child, the leaf node in the sorting tree consists of just one response. In the case of an agent child, the leaf node consists of an already sorted list of servers available in that child's sub-hierarchy. For small values of N , the sorting overhead incurred by an agent is thus more strongly related to the number of direct children the node has than to the number of SeDs included in the deep sub-hierarchy below the agent. Increasing the number of children an agent has increases the agent's sorting time while increasing the depth of the agent hierarchy increases the communication latency incurred during the hierarchical decision process.

While the agent aggregation routines are designed to select the best servers for a problem, it is in fact even more important that they ensure a decision is always made. The sorting approach thus relies on a series of comparison options where each comparison level utilizes a different type of SeD-provided data. In this way, the agent hierarchy does not become deadlocked simply because, for example, some of the SeDs do not have the capability of providing an application-specific performance prediction. In fact, for system stability, any agent-level sorting routine should rely on a final random selection option to provide a last-resort option for choosing between servers.

5.2 Scheduling Extensions

The distributed approach chosen in the DIET platform allows the study of other algorithms where some intelligence can be put at various levels of the hierarchy. One first optimization consists in adding queue-like semantics to the DIET server and master agent levels (Dail and Desprez 2005).

At the server level, the number of concurrent jobs allowed on a server can be limited. This control can greatly improve performance for resource-intensive applications where resource sharing can be very harmful to performance. Such control at the server-level is also necessary to support some distributed scheduling approaches of interest. The following paragraphs show, through an example, which kinds of problems may appear.

As a simple first approach we do not attempt to keep extra jobs from reaching the SeD. Instead, once solve requests reach the SeD we place their threads in what we will call an **SeD-level queue**. In fact, to keep overheads low we implement a very lightweight approach that offers some, but not all, of the semantics of a full queue. We add a counting semaphore to the SeD and initialize the semaphore with a user-configurable value defining the desired limit on concurrent solves. When each request finishes its computational work, it calls a post on the counting semaphore to allow another request to begin computing. The order in which processes will be woken up while waiting on a semaphore is not guaranteed on many systems; therefore we augmented the semaphore to ensure that threads are released in the appropriate order. Figure 4 provides an overview of the queueing structures added.

To support consideration of queue effects in the scheduling process, we use a number of approaches for tracking queue statistics. It is not possible to have complete information on all the jobs in the queue without adding significant overhead for coordinating the storage of queue data between all requests. Thus we approximate queue statistics by storing the number of jobs waiting in the queue and the sum of the predicted execution times for all waiting jobs. Once jobs begin executing we individually store and track



Fig. 4 DIET extensions for request flow control.

the jobs' predicted completion time. By combining these data metrics and taking into account the number of physical processors and the user defined limit on concurrent solves, we can provide a rough estimate of when a new job would begin execution. This estimate is included by the SeD with the other performance estimates passed up the hierarchy during a schedule request.

There are some disadvantages to this method of controlling request flow. Most importantly, requests are in fact resident on the server while they wait for permission to begin their solve phase. Thus, if the parameters of the problem sent in the solve phase include large data sets, memory-usage or disk-usage conflicts could be seen between the running jobs and the waiting requests. Some DIET applications with very large data sets use a different approach for transferring their data where only the file location (e.g. perhaps an http locator for publicly available data) is sent in the problem parameters and the data is retrieved at the beginning of the solve. The impact of this problem will therefore depend on the data approach used by the application. A second problem with

this approach arises from the fact that once requests are allocated to a particular server, DIET does not currently support movement of the request to a different server. When system conditions change, although the jobs have not begun executing, DIET cannot adjust the placement to adapt to the new situation. Thus performance will suffer in cases of unexpected competing load or poorly predicted job execution time. Also, in the case of improvements in the system, such as the dynamic addition of server resources, DIET cannot take advantage of the resources for those tasks already allocated to servers. To avoid this problem we could plan to integrate the ability to carry out task migrations. The last, but not least, problem relates to fault-tolerance. If a server crashes, we lose the queue information. Thus a replication mechanism should be implemented.

At the master agent level, under high-load conditions, incoming requests can be stalled at the master agent and then scheduled as a batch at an appropriate time. This batch window addition can be used to test a variety of scheduling approaches: the MA can re-order tasks to accommodate

data dependencies, co-scheduling of multiple tasks on the same resource can be avoided even when the requests arrive nearly simultaneously, and inter-task dependencies can be accounted for in the scheduling process. In the standard DIET system, requests are each assigned an independent thread in the master agent process and that thread persists until the request has been forwarded in the DIET hierarchy, the response received, and the final response forwarded on to the user. In this approach, the only data object shared among threads is a counter that is used to assign a unique request ID to every request. In the modified master agent, each request is still assigned a thread that persists until the response has been sent back to the client. However, we introduce one additional thread that provides higher-level management of request flow. Scheduling proceeds in distinct phases called windows and both the number of requests scheduled in a window and the time interval spent between windows are configurable. An interesting aspect of this algorithm is that the master agent can only discern characteristics of the DIET hierarchy, such as server availability, by forwarding a request in the hierarchy. We avoid sending any task twice in the hierarchy, thus the Global-TaskManager must schedule some jobs in order to have information about server loads and queue lengths. Information may be given from the NWS sensor (Wolski, Spring, and Hayes 1999), as discussed in Section 4.

5.3 Plug-in Schedulers

Finally, we are now working on plug-in schedulers specially designed for expert users who wish to upgrade the scheduling for a specific application. This will allow the user to play with the internals of agents and tune DIET's scheduling by changing the heuristics, adding queues, changing the performance metrics and the aggregation functions, etc. Also we believe that this feature will be useful both for computer scientists to test their algorithms on a real platform and for expert application scientists to tune DIET for specific application behavior.

6 Data Management

GridRPC environments such as NetSolve, Ninf, and DIET are based on the client-server programming paradigm. However, generally in this paradigm, no data management is performed. As in the standard RPC model, request parameters (input and output data) are sent back and forth between the client and the remote server. Data is not supposed to be available on a server for another step of the algorithm (a new RPC) once a step is finished. This drawback can lead to extra overhead as a result of useless communications over the net.

This problem has been identified by the NetSolve and Ninf projects as a major performance loss. NetSolve has

proposed several ways to keep data in place. The first approach is called *request sequencing* (Arnold, Bachmann, and Dongarra 2000). It consists of scheduling a sequence of NetSolve calls on one server. The sequence of requests written between two sequence delimiters `netssl_sequence_begin` and `netssl_sequence_start` is analyzed and a dataflow graph is computed that allows useless data transfers to be avoided. However, this feature is only available on a single server without redistribution between servers. Another approach is called *Distributed Storage Infrastructure* (DSI; Arnold, Casanova, and Dongarra 2002). The DSI helps the user to control the placement of data that will be accessed by a server. Instead of having multiple transmissions of the same data, DSI allows the transfer of the data once from the client to a storage server. A data handle is then used at the request level. DSI acts as a data cache. One instance of a DSI is based on IBP (Internet Backplane Protocol)¹. This approach is interesting, but is not connected to the choice of computational servers. A last optimization has been provided that allows the redistribution of the data between servers and the persistence of data (Desprez and Jeannot 2004). A new API is provided that allows a client to manage its data locally and remotely between request calls. Ninf has similar solutions with other data management systems.

6.1 Data Tree Manager

A first data management service has been developed for the DIET platform (Del Fabbro et al. 2005) called the Data Tree Manager (DTM). This DIET data management model is based on two key elements: the data identifiers and the DTM. To avoid multiple transmissions of the same data from a client to a server, the DTM allows data to be left inside the platform after computation while data identifiers will be used further by the client to reference its data.

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

In order to avoid interlacing between data messages and computation messages, the proposed architecture separates data management from computation management. The DTM is built around three entities, the logical data manager, the physical data manager, and the data mover (see Figures 5 and 6).

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

Table 1
Persistence Modes

Mode	Description
DIET_VOLATILE	not stored
DIET_PERSISTENT_RETURN	stored on server, movable and copy back to client
DIET_PERSISTENT	stored on server and movable
DIET_STICKY	stored and non movable
DIET_STICKY_RETURN	stored, non movable and copy back to client

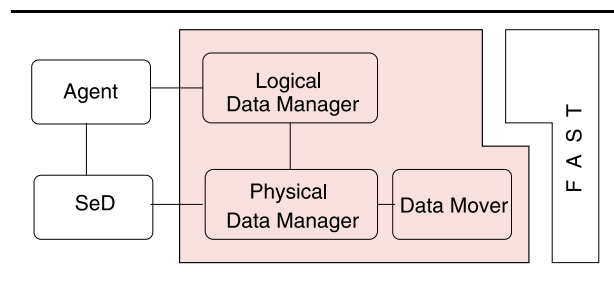


Fig. 5 DTM: Data Tree Manager.

The Physical Data Manager is composed of a set of DataManager objects. The DataManager is located onto

each SeD with which it communicates locally. It owns a list of persistent data. It stores data and provides data to the server when needed. It provides features for data movement and it informs its LocManager parent of updating operations performed on its data (add, move, delete). Moreover, if a data is duplicated from a server to another one, the copy is set as non-persistent and destroyed after its use with no hierarchy update.

This structure is built in a hierarchical way as shown in Figure 6. It is mapped on the DIET architecture. There are several advantages of defining such a hierarchy. First, communications between agents (MA or LA) and data location objects (LocManager) are local as are those between computational servers (SeD) and data storage objects (DataManager). This ensures a lower cost for the communication

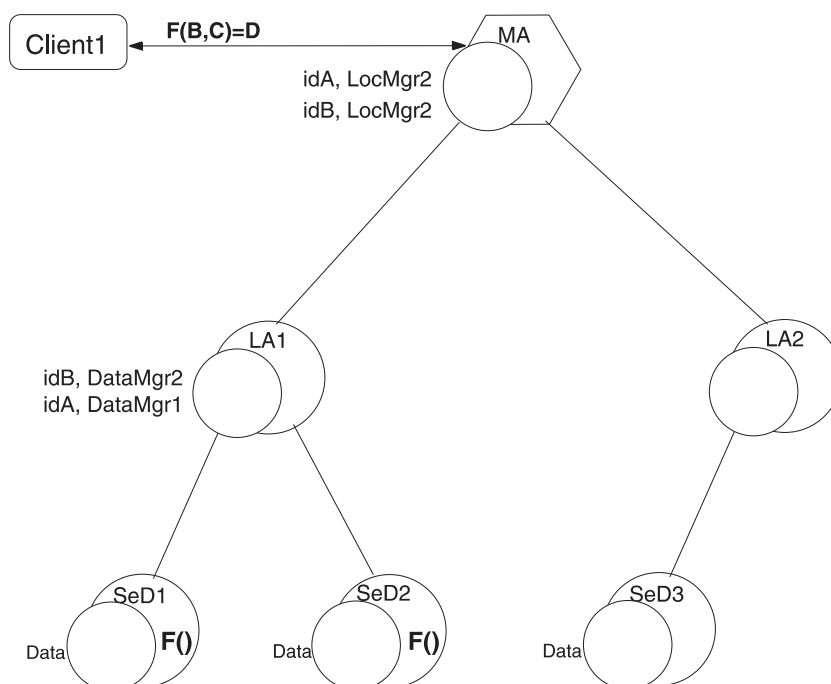


Fig. 6 DataManager and LocManager objects.

needed for agents to get information on data location and for servers to retrieve data. Secondly, considering the physical repartition of the architecture nodes (an LA front-end of a local area network for example), when data transfers between servers localized in the same subtree occur, the following updates are limited to this subtree. So, the rest of the platform is not involved in the updates.

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

6.2 JuxMem

JuxMem (Juxtaposed Memory; Antoniu, Bougé, and Jan 2003) is a peer-to-peer architecture which provides a memory sharing service allowing peers to share memory data as well as files (Note that from the DIET viewpoint memory sharing or file sharing have similar behavior). The software architecture of JuxMem, mirrors a hardware architecture consisting of a federation of distributed clusters and is therefore *hierarchical*. The JuxMem architecture is made up of a network of peer groups, which generally correspond to clusters at the physical level. All the groups are inside a wider group which includes all the peers which run the service (the JuxMem group). Each cluster group

consists of a set of nodes which provide memory for data storage. We will call these nodes *providers*. In each cluster group, a node is used to make up the backbone of JuxMem's network of peers. This node is called *cluster manager*. Finally, a node which simply uses the service to allocate and/or access data blocks is called *client*. It should be stressed that a node may at the same time act as a cluster manager, a client, and a provider. However, for the sake of clarity, each node only plays a single role in the example illustrated in Figure 7.

Each block of data stored in the system is associated with a group of peers called a data group. Note that a data group can be made up of providers from different cluster groups. Indeed, data can be spread over several clusters. For this reason, the data and cluster groups are at the same level of the group hierarchy. Note that the cluster groups could also correspond to subsets of the same physical cluster.

Another important feature is that the architecture of JuxMem is dynamic, since clusters and data groups can be created at run time. For instance, a data group is automatically instantiated for each block of data inserted into the system.

The integration of JuxMem with DIET can be done in two modes: sharing and concurrent model. The sharing model is the solution invoked by the principle of grid offered for a large number of resources. The resources with

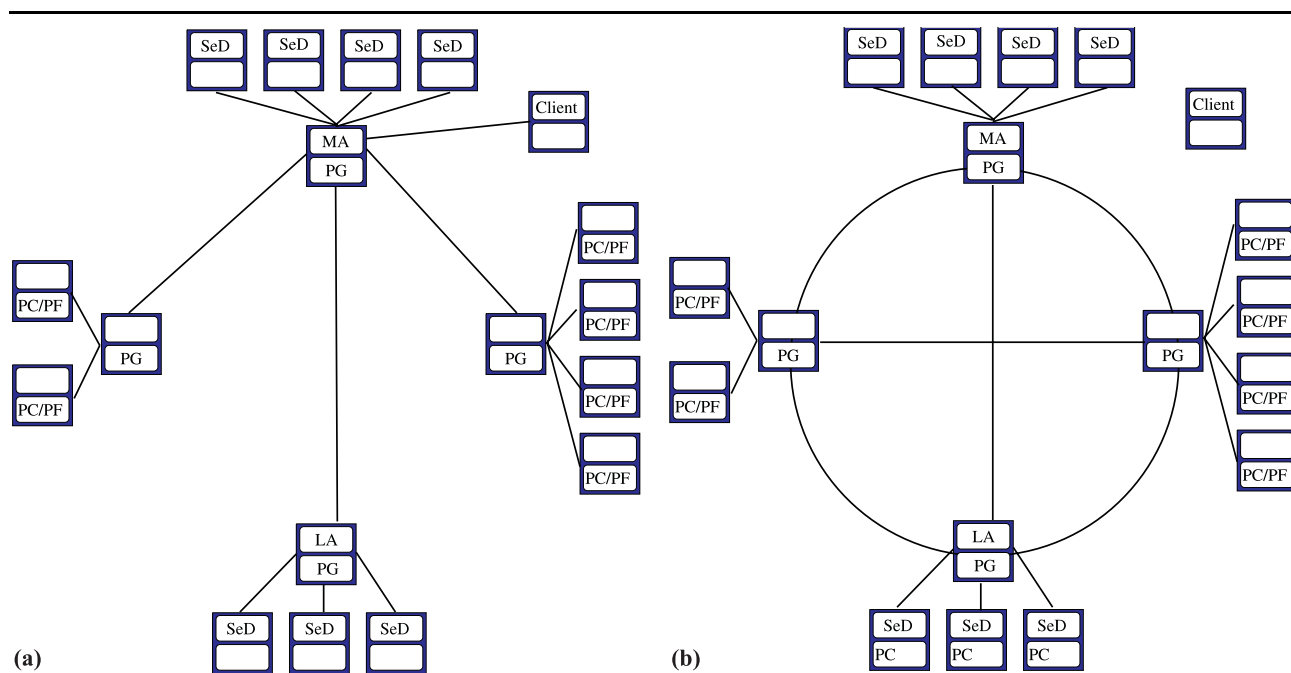


Fig. 7 (a) Sharing model. (b) Concurrent model.

different attributes can be DIET components or JuxMem components as shown in Figure 7 (a). The advantage of this model is that it limits the interference between the two system foundations to utilize the memory shared by others. On the other hand the number of resources for each system is reduced.

The concurrent model, shown in Figure 7 (b), allows sharing of the grid resources between DIET and JuxMem, but this model increases the difficulty of having a correct outcome for the performance forecasting tool. Indeed, in pursuit of simplicity, we avoid this model because there is no mechanism to communicate between DIET and JuxMem to take into account the impact of one on the other.

7 Deployment and Monitoring of the Platform

This section focuses on the deployment of DIET. Although the deployment of such an architecture may be constrained, e.g. by a firewall, access rights or security, its efficiency depends heavily on the quality of the mapping between its different components and the grid resources. In Caron, Chouhan, and Legrand (2004) we have proposed a new model based on linear programming to estimate the performance of a deployment of a hierarchical set of schedulers. The advantages of our modeling approach are: evaluation of a virtual deployment before a real deployment; provision of a decision builder tool (i.e. designed to compare different architectures or to add new resources); and taking into account scalability of the platform. Using our model, it is possible to determine the bottleneck of the platform and thus to know whether a given deployment can be improved or not.

7.1 Steady-state Scheduling

A collection of heterogeneous resources (e.g. a processor or a cluster) and the communication links between them is naturally modeled as nodes and edges of an undirected tree-shaped graph. Each node is a computing resource capable of computing and communicating with its neighbors at different rates. We assume that one specific node, referred as the client, initially generates requests and floods the MAs with these requests. The main problem is then to determine a steady state scheduling policy for each processor, i.e. the relative proportions of time spent: computing the request coming from client to server; selecting the best server; sending the request; and receiving the reply packet (reply of the request). This is necessary so that the (average) overall number of requests processed at each time-step can be maximized.

Beaumont et al. (2002) solved the steady-state master-slave scheduling problem for a tree-shaped heterogeneous platform. They explained how to allocate a large number

of independent and equal size tasks on a heterogeneous grid computing platform. They first computed the maximum steady-state throughput of the platform using a linear program. Then, they showed that this throughput could be reached if each node locally used a bandwidth-centric strategy which states that: if enough bandwidth is available, then all children nodes are kept busy; if bandwidth is limited, then tasks should be allocated first to the children which have sufficiently small communication times, regardless of their computation power.

Some interesting points of this theoretical framework, such as the steady-state scheduling strategy, equal size of requests, using linear constraints, etc., can be applied to our practical framework.

7.2 Hierarchical Deployment Model

Here we describe how we model the deployment problem. It is beyond the scope of this paper to provide the complete model. For more details the reader can refer to Caron, Chouhan, and Legrand (2004).

The target platform is represented by a weighted graph $G = (V, E, w, c)$. Each $P_i \in V$ represents a computing resource of computing power w_i , meaning that node P_i executes w_i MFlop/second (so the bigger the w_i , the faster the computing resource P_i). There is a client node, i.e. a node P_c , which generates the requests that are passed to the following nodes². Each link $P_i \rightarrow P_j$ is labeled by the bandwidth value $c_{i,j}$ which represents the size of data sent per second between P_i and P_j . The unit used for link bandwidth is Mb/second. The size of the request generated by the client is $S_i^{(in)}$ and the size of the reply request created by each node is $S_o^{(out)}$. The unit used for these quantities is Mb/request. The amount of computation needed by P_i to process one incoming request is denoted by $W_i^{(in)}$ and the amount of computation needed by P_i to merge the reply requests of its children is denoted by $W_o^{(out)}$. We denote by $W_D^{(DGEMM)}$ the amount of computation needed by P_i to process a generic problem (i.e., level 3 BLAS matrix multiplication function called DGEMM). We selected a BLAS routine as it gives good forecast predictions (Caron and Suter 2002) and can be easily expressed as linear constraints.

7.3 Automatic Deployment and Redeployment

Even when neglecting the servers' constraints, finding the best topology is a hard problem since it amounts to finding the best broadcast tree on a general graph, which is known to be NP-complete (Beaumont et al. 2003). Note that even when neglecting the request mechanism, as soon as one takes in account the communications of the problem's data, the problem of finding the best deployment becomes NP-complete too (Banino et al. 2002).

```

1: while (number of available nodes > 0) do
2:   Calculate the throughput  $\rho$  of structure.
3:   Find a node whose constraint is tight and
     that can be split
4:   if no such node exists then
5:     The deployment cannot be improved.
6:     Exit
   endif
7:   Split the load by adding new node to its parent
8:   Decrease the number of available nodes
endwhile

```

Algorithm 1 Algorithm to add an LA.

Nevertheless, in real life, the topology of the underlying platform is particular and enforces some parts of the deployment. Therefore, we propose to improve the throughput of a given deployment by removing its bottleneck. Using the previous theorems, we can find the bottlenecks and get rid of them by adding more LAs to the parent of a loaded LA so as to divide the load of that particular LA. We add new LAs according to the greedy Algorithm 1.

7.4 Configuration and Launch

In work complementary to the previous theoretical approach, we developed GoDIET which is a tool for the configuration, launching, and management of DIET on computational grids. Users of GoDIET write an XML file describing their available compute and storage resources and the desired overlay of DIET agents and servers onto those resources. GoDIET automatically generates and stages all necessary configuration files, launches agents and servers in appropriate hierarchical order, reports feedback on the status of running components, and allows shutdown of all launched software.

7.5 Associated Services

A number of associated services can optionally be used in conjunction with DIET. Since DIET uses CORBA for all communication activities, DIET can directly benefit from the **CORBA naming service** – a service for the mapping of string-based names for objects to their localization information. For example, the MA is assigned a name in the MA configuration file; then, during startup, the MA registers with the naming service by providing this string-based name as well as all information necessary for other components to communication with the MA (e.g. machine hostname and port). When another component such as an LA needs to contact the MA, the LA uses the string-based

name to lookup contact information for the MA. Therefore, in order to register with the DIET hierarchy, a DIET element need only have (1) the host and port on which the naming service can be found and (2) the string-based name for the element's parent.

DIET also uses a CORBA-based logging service called **LogService**, a software package that provides interfaces for generation and sending of log messages by distributed components, a centralized service that collects and organizes all log messages, and the ability to connect any number of listening tools to which LogService will send all or a filtered set of log messages. LogService is robust against failures of both senders of log messages and listeners for log updates. When LogService usage is enabled in DIET, all agents and SeDs send log messages indicating their existence and a special configurable field is used to indicate the name of the element's parent. Messages can also be sent to trace requests through the system or to monitor resource performance (e.g. CPU availability on a particular SeD's host or the network bandwidth between an agent and an SeD connected to the agent).

VizDIET is a tool that provides a graphical view of the DIET deployment and detailed statistical analysis of a variety of platform characteristics such as the performance of request scheduling and solves. To provide real-time analysis and monitoring of a running DIET platform, VizDIET can register as a listener to LogService and thus receives all platform updates as log messages sent via CORBA. Alternatively, to perform visualization and processing post-mortem, VizDIET uses a static log message file that is generated during run-time by LogService and set aside for later analysis. Figure 8 presents a screenshot of VizDIET.

7.6 GoDIET

The goal of GoDIET is to automate the deployment of DIET platforms and associated services for diverse grid environments. Specifically, GoDIET automatically generates configuration files for each DIET component taking into account user configuration preferences and the hierarchy defined by the user, launches complementary 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. Figure 9 provides an overview of the interactions between a running DIET platform, LogService, and VizDIET. We now describe the third external service in the figure – GoDIET.

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; we have chosen Java for the



Fig. 8 VizDIET Screenshot.

GoDIET implementation as it satisfies all of these requirements and provides for rapid prototyping. The description of resources, the software to deploy, and user preferences are defined in an XML file; we use a Document Type Definition file (DTD) to provide automated enforcement of an allowed XML file structure.

More specifically, the GoDIET XML file contains the description of DIET agents and servers and their hierarchy, the description of desired complementary services such as LogService, the physical machines to be used, the disk space available on these machines, and the configuration of paths for the location of needed binaries and dynamically loadable libraries. The file format provides a strict separation of the resource description and the deployment configuration description; the resource description portion must be written once for each new grid environment, but can then be re-used for a variety of deployment configurations.

The basic user interface is a non-graphical console mode and can be used on any machine where Java is available and where the machine has *ssh* access to the target resources used in the deployment. An alternative interface is a graphical console that can be loaded by VizDIET to provide an integrated management and visualization tool. Both the graphical and non-graphical console modes can report a variety of information on the deployment including the run status and, if running, the PID of each component, as well as whether log feedback has been obtained for each component. GoDIET can also be launched in *batch* mode where the platform can be launched and stopped without user interaction; this mode is primarily useful for experiments.

We use *scp* and *ssh* to provide secure file transfer and task execution; *ssh* is a tool for remote machine access that has become (nearly) universally available on grid resources in recent years. With a carefully configured *ssh* com-

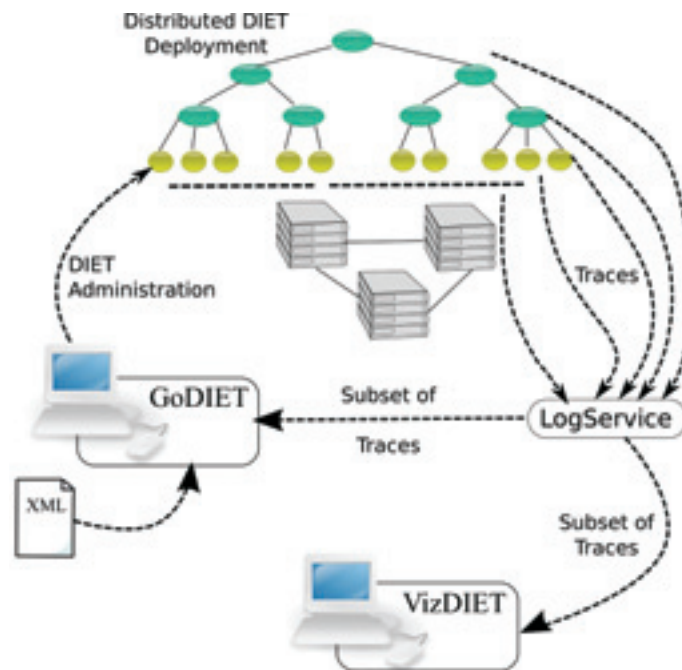


Fig. 9 Interaction of GoDIET, LogService, and VizDIET to assist users in controlling and understanding DIET platforms.

mand, GoDIET can configure environment variables, specify the binary to launch with appropriate command line parameters, and specify different files for the stdout and stderr of the launched process. Additionally, for a successful launch, GoDIET can retrieve the PID of the launched process; this PID can be used later for shutting down the DIET deployment. In the case of a failure to launch the process, GoDIET can retrieve these messages and provide them to the user. To illustrate the approach used, an example of the type of command used by GoDIET follows:

```
/bin/sh -c ( /bin/echo %<
"export PATH=/home/user/local/bin/.$PATH ; %<
export LD_LIBRARY_PATH=/home/user/local/lib ; %<
export OMNIORB_CONFIG=/home/user/godiet_s/
run_04Jul01/omniORB4.cfg; %<
cd /home/user/godiet_s/run_04Jul01; %<
nohup dietAgent ./MA_0.cfg < /dev/null > MA_0.out
2> MA_0.err &" ; %<
/bin/echo '/bin/echo ${!}' ) %<
| /usr/bin/ssh -q user@ls2.ens.vthd.prd.fr /
bin/sh - %<
```

It is important that each `ssh` connection can be closed, once the launch is complete, while leaving the remote process running. If this cannot be achieved, the system will eventually run out of resources (typically sockets) and refuse to open additional connections. In order to enable a scalable launch process, the above command ensures that the `ssh` connection can be closed after the process is launched. Specifically, in order for this connection to be closeable: (1) the UNIX command `nohup` is necessary to ensure that when the connection is closed the launched process is not killed as well; (2) the process must be put in the background after launch; and (3) redirection of all inputs and outputs for the process is required.

7.7 DIET Deployment

The DIET platform is constructed following the hierarchy of agents and SeDs. The very first element to be launched during deployment is the naming service; all other elements are provided with the hostname and port at which the naming service can be found. Afterwards, deployed elements can locate other elements on the grid using only

the element's string-based name and the contact information for the naming service. After the naming service, the MA is launched; the MA is the root of the DIET hierarchy and thus does not register with any other elements. After the MA, all other DIET elements understand their place in the hierarchy from their configuration file which contains the name of the element's parent. Users of GoDIET specify the desired hierarchy in a more intuitive way via the naturally hierarchical XML input file to GoDIET. Based on the user-specified hierarchy, GoDIET automatically generates the appropriate configuration files for each element, and launches elements in a top-down fashion to respect the hierarchical organization.

As a benefit of this approach, multiple DIET deployments can be launched on the same group of machines without conflict as long as the name services for each deployment uses a different port and/or a different machine.

DIET provides the features and flexibility to allow a wide variety of deployment configurations, even in difficult network and firewall environments. For example, for platforms without DNS-based name resolution or for machines with both private and public network interfaces, elements can be manually assigned an *endpoint* hostname or IP in their configuration files; when the element registers with the naming service, it specifically requests this endpoint be given as the contact address during name lookups. Similarly, an *endpoint* port can be defined to provide for situations with limited open ports in firewalls. These specialized options are provided to DIET elements at launch time via their configuration files; GoDIET supports these configuration options via more user-intuitive options in the input XML file and then automatically incorporates the appropriate options while generating each element's configuration file. For large deployments, it is key to have a tool like GoDIET to make practical use of these features.

8 Related Work

Several other Network Enabled Server systems have been developed in the past (Arbenz, Gander, and Mori 1997; Ferris, Mesnier, and Mori 2000; Matsuoka et al. 2000; NEOS). Among them, NetSolve (Arnold et al. 2001) and Ninf (Nakada, Sato, and Sekiguchi 1999; <http://ninf.apgrid.org/>) have further advanced research around the GridRPC paradigm.

NetSolve (Arnold et al. 2001) has been developed at the University of Tennessee, Knoxville. NetSolve allows the connection of clients (written in C, C++, Fortran, Matlab, etc.) to solve requests sent to servers found by a single agent. This centralized agent maintains a list of available servers along with their capabilities. Servers are sent information about their status at a given frequency. Scheduling is done based on simple models provided by the application developers, LINPACK benchmarks executed on remote

servers, and information given by NWS. Some fault tolerance is also provided at the agent level. Data management is also done either through request sequencing or using IBP (see Section 6). Security is also addressed using Kerberos. Client Proxies ensure a good portability and interoperability with other systems such as Ninf or Globus (Arnold, Casanova, and Dongarra 2002). The NetSolve team has recently introduced GrADSolve (Vadhiyar and Dongarra 2004), an RPC system based on the GrADS architecture. This new framework allows the dynamic choice of resources taking into account application and resource properties.

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 (Tanaka et al. 2003; 2005) or Web Services (Shirasuna et al. 2002). Fault tolerance is also provided using Condor and a checkpointing library (Nakada et al. 2004). The performance of the platform can be studied using a powerful tool called BRICKS.

The main differences between the NES systems presented in this section and DIET are: the use of distributed scheduling in DIET that allows a better scalability when the number of clients is large; and the request frequency is high and the use of CORBA as a middleware in DIET.

9 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 with NetSolve and Ninf, DIET provides an interface to the GridRPC API defined within the Global Grid Forum.

Our main objective was 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 were 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). Data management is also an important part of the performance gain when dependences exist between requests. We investigated two approaches. One related to the DIET architecture (DTM) and one that acts as a data cache (JuxMem). The management of the platform was handled by several tools like GoDIET for the automatic deployment of the different components, LogService for the monitoring, and VizDIET for the visualization of the behavior of the DIET's internals.

Many applications have also been ported on DIET in chemical engineering, physics, bioinformatics, robotics, etc.

Our future work will focus on adding more flexibility using plug-in schedulers, improving the dynamicity of the

platform using P2P connection (with JXTA), improving the relations between the schedulers and the data managers, and finally validating the whole platform at a large scale within the GRID'5000 project (<http://www.grid5000.org/>).

Acknowledgements

The authors wish to thank all our colleagues who contributed to the design and development of DIET and related tools. In particular, we would like to thank R. Bolze, M. Boury, Y. Caniou, P. Combes, P. Kaur Chouhan, S. Dahan, H. Dail, B. DelFabbro, G. Hoesh, E. Jeannot, A. Legrand, F. Lombard, J.-M. Nicod, C. Pera, F. Petit, L. Philippe, C. Pontvieux, M. Quinson, A. Su, F. Suter, C. Tedeschi, and A. Vernois. We also want to thank our colleagues porting applications on top of DIET and in particular, J.-L. Barrat, C. Blanchet, F. Boughmar, M. Dayde, C. Hamerling, G. Monard, R. Sommet, and S. Vialle.

Author Biographies

Eddy Caron is Assistant Professor at Ecole Normale Supérieure de Lyon and holds a position at LIP laboratory (ENS Lyon, France). He is a member of GRAAL project and technical manager for the DIET software. He received his PhD in C.S. from University de Picardie Jules Verne in 2000. His research interests include parallel libraries for scientific computing on parallel distributed memory machines, problem solving environments, and grid computing. See <http://graal.ens-lyon.fr/~ecaron> for further information.

Frédéric Desprez is a director of research at INRIA and holds a position at LIP laboratory (ENS Lyon, France). He received his PhD in C.S. from the Institut National Polytechnique de Grenoble in 1994 and his MS in C.S. from the ENS Lyon in 1990. His research interests include parallel libraries for scientific computing on parallel distributed memory machines, problem solving environments, and grid computing. See <http://graal.ens-lyon.fr/~desprez> for further information.

Notes

- 1 <http://loci.cs.utk.edu/>
- 2 We use only one client node for the sake of simplicity but modeling many different clients with different problem types can be done easily.

References

Antoniou, G., Bougé, L., and Jan, M. 2003. JuxMem: An adaptive supportive platform for data sharing on the grid. *Proceedings of Workshop on Adaptive Grid Middleware*

(*AGRIDM 2003*), pp. 49–59, New Orleans, Louisiana. Held in conjunction with PACT 2003. Extended version to appear in *Kluwer Journal of Supercomputing*.

- Arbenz, P., Gander, W., and Mori, J. 1997. The Remote Computational System. *Parallel Computing* 23(10):1421–1428.
- Arnold, D., Agrawal, S., Blackford, S., Dongarra, J., Miller, M., Sagi, K., Shi, Z., and Vadhiyar, S. 2001. Users' Guide to NetSolve V1.4. Computer Science Dept. Technical Report CS-01-467, University of Tennessee, Knoxville, TN. <http://www.cs.utk.edu/netsolve/>.
- Arnold, D. C., Bachmann, D., and Dongarra, J. 2000. Request Sequencing: Optimizing Communication for the Grid. *Euro-Par 2000 Parallel Processing, 6th International Euro-Par Conference*, volume 1900 of Lecture Notes in Computer Science, pp. 1213–1222, Munich: Springer Verlag.
- Arnold, D. C., Casanova, H., and Dongarra, J. 2002. Innovations of the NetSolve Grid Computing System. *Concurrency And Computation: Practice And Experience* 14:1–23.
- Banino, C., Beaumont, O., Legrand, A., and Robert, Y. 2002. Scheduling strategies for master-slave tasking on heterogeneous processor grids. *PARA'02: International Conference on Applied Parallel Computing*, LNCS 2367. Munich: Springer Verlag.
- Beaumont, O., Carter, L., Ferrante, J., Legrand, A., and Robert, Y. 2002. Bandwidth-Centric Allocation of Independent Tasks on Heterogeneous Platforms. *International Parallel and Distributed Processing Symposium IPDPS'2002*. Los Alamitos, CA: IEEE Computer Society Press.
- Beaumont, O., Legrand, A., Marchal, L., and Robert, Y. 2003. Optimizing the steady-state throughput of broadcasts on heterogeneous platforms. Technical Report 2003-34, LIP.
- Berman, F., Fox, G., and Hey, A. (Eds) 2003. *Grid Computing: Making the Global Infrastructure a Reality*. New York: Wiley.
- Caron, E., Desprez, F., Petit, F., and Tedeschi, C. 2005. A Peer-to-Peer Extension of Network-Enabled Server Systems. In *e-Science 2005. First IEEE International Conference on e-Science and Grid Computing*, Melbourne, Australia, pages 430–437, 5–8 December.
- Caron, E., Desprez, F., Lombard, F., Nicod, J.-M., Quinson, M., and Suter, F. 2002. A Scalable Approach to Network Enabled Servers. *Proceedings of EuroPar 2002*, Paderborn, Germany.
- Caron, E., Desprez, F., Petit, F., and Tedeschi, C. 2004. Resource Localization Using Peer-To-Peer Technology for Network Enabled Servers. Research report 2004-55, Laboratoire de l'Informatique du Parallélisme (LIP).
- Caron, E. and Suter, F. 2002. Parallel Extension of a Dynamic Performance Forecasting Tool. *Proceedings of the International Symposium on Parallel and Distributed Computing*, pp. 80–93, Iasi, Romania.
- Dail, H. and Desprez, F. 2005. Experiences with Hierarchical Request Flow Management for Network Enabled Server Environments. Technical Report TR-2005-07, LIP ENS Lyon.
- Del Fabbro, B., Laiymani, D., Nicod, J.-M., and Philippe, L. 2005. Data management in grid applications providers. *IEEE International Conference DFMA'05*, Besançon, France. to appear.

- Denis, A., Perez, C., and Priol, T. 2001. Towards high performance CORBA and MPI middlewares for grid computing. Lee, C. A., ed, *Proceedings of the 2nd International Workshop on Grid Computing*, number 2242 in LNCS, pp. 14–25, Denver, Colorado, USA. Munich: Springer Verlag.
- Desprez, F. and Jeannot, E. 2004. Improving the GridRPC Model with Data Persistence and Redistribution. *3rd International Symposium on Parallel and Distributed Computing (ISPDC)*, Cork, Ireland.
- Desprez, F., Quinson, M., and Suter, F. 2001. Dynamic Performance Forecasting for Network Enabled Servers in a Metacomputing Environment. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001)*.
- Ferris, M., Mesnier, M., and Mori, J. 2000. NEOS and Condor: Solving Optimization Problems Over the Internet. *ACM Transaction on Mathematical Software* 26(1):1–18. <http://www-unix.mcs.anl.gov/metaneos/publications/index.html>.
- Foster, I. and Kesselman, C. (Eds) 2004. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann.
- Good, M. and Goux, J.-P. 2000. iMW: A web-based problem solving environment for grid computing applications. Technical report, Department of Electrical and Computer Engineering, Northwestern University.
- Haupt, T., Akarsu, E., and Fox, G. 2000. WebFlow: A Framework for Web Based Metacomputing. *Future Generation Computer Systems* 16(5):445–451.
- Henning, M. and Vinoski, S. 1999. *Advanced CORBA(R) Programming with C++*. Addison-Wesley Pub Co.
- Jan, M. and Noblet, D. 2004. Performance Evaluation of JXTA Communication Layers. Technical Report RR-5530, INRIA, IRISA, Rennes, France.
- Kapadia, N., Robertson, J., and Fortes, J. 1998. Interfaces Issues in Running Computer Architecture Tools via the World Wide Web. *Workshop on Computer Architecture Education at ISCA 1998*, Barcelona. <http://www.ecn.purdue.edu/labs/punch/>.
- Keller, R., Krammer, B., Mueller, M., Resch, M. M., and Gabriel, E. 2003. MPI Development Tools and Applications for the Grid. *Workshop on Grid Applications and Programming Tools*, held in conjunction with the GGF8 meetings, Seattle.
- Matsuoka, S. and Casanova, H. 2000. Network-Enabled Server Systems and the Computational Grid. Grid Forum, Advanced Programming Models Working Group whitepaper (draft).
- Matsuoka, S., Nakada, H., Sato, M., and Sekiguchi, S. 2000. Design Issues of Network Enabled Server Systems for the Grid. Grid Forum, Advanced Programming Models Working Group whitepaper.
- MPICH-G. <http://www.hpclab.niu.edu/mpi/>.
- Nakada, H., Sato, M., and Sekiguchi, S. 1999. Design and Implementations of Ninf: towards a Global Computing Infrastructure. *Future Generation Computing Systems, Metacomputing Issue* 15(5-6):649–658. <http://ninf.apgrid.org/papers/papers.shtml>.
- Nakada, H., Tanaka, Y., Matsuoka, S., and Sekiguchi, S. 2004. The Design and Implementation of a Fault-Tolerant RPC System: Ninf-C. *Proceedings of HPC Asia 2004*, pp. 9–18.
- NEOS. NEOS Server. <http://www-neos.mcs.anl.gov/>.
- Quinson, M. 2002. Dynamic Performance Forecasting for Network-Enabled Servers in a Metacomputing Environment. *International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS'02)*, in conjunction with IPDPS'02.
- Seymour, K., Lee, C., Desprez, F., Nakada, H., and Tanaka, Y. 2004. The End-User and Middleware APIs for GridRPC. *Workshop on Grid Application Programming Interfaces*, in conjunction with GGF12, Brussels, Belgium.
- Shirasuna, S., Nakada, H., Matsuoka, S., and Sekiguchi, S. 2002. Evaluating Web Services Based Implementations of GridRPC. *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11 2002)*, pp. 237–245. <http://matsu-www.is.titech.ac.jp/sirasuna/research/hpdc2002/hpdc2002.pdf>.
- Sun Microsystems Inc. Sun grid engine. <http://www.sun.com/gridware/>.
- Tanaka, Y., Nakada, N., Sekiguchi, S., Suzumura, T., and Matsuoka, S. 2003. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *Journal of Grid Computing* 1:41–51.
- Tanaka, Y., Takemiya, H., Nakada, H., and Sekiguchi, S. 2005. Design, Implementation and Performance Evaluation of GridRPC Programming Middleware for a Large-Scale Computational Grid. *Proceedings of 5th IEEE/ACM International Workshop on Grid Computing*, pp. 298–305.
- Thain, D., Tannenbaum, T., and Livny, M. 2004. Distributed Computing in Practice: The Condor Experience. *Concurrency and Computation: Practice and Experience* 17(2–4): 323–356.
- Vadhiyar, S. and Dongarra, J. 2004. GrADSolve – A grid-based RPC System for Parallel Computing with Application Level Scheduling. *Journal of Parallel and Distributed Computing* 64:774–783.
- Wolski, R., Spring, N. T., and Hayes, J. 1999. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computing Systems, Metacomputing Issue* 15(5–6):757–768.