Contribution to the management of large scale platforms: the Diet experience

Eddy Caron

HABILITATION A DIRIGER DES RECHERCHES

présentée par

Eddy Caron

pour obtenir

l'habilitation à diriger des recherches

de l'**École Normale Supérieure de Lyon**

spécialité : Informatique

Contribution to the management of large scale platforms: the DIET experience

---

Date de soutenance: 6 octobre 2010

Composition du Jury :	*Rapporteurs*	: Dick	H. J. Epema (Technische Universiteit Delft)
		: Thierry	Priol (INRIA. IRISA. Rennes)
		: Pierre	Sens (INRIA. LIP6. Paris)
	Examinateurs	: Vincent	Breton (CNRS. LPC. Clermont Ferrand)
		: Franck	Cappello (INRIA. Urbana Champaign University)
		: Olivier	Richard (IMAG. Grenoble)
	Directeur	: Frédéric	Desprez (INRIA. ENS-Lyon)

Habilitation préparée au sein du
Laboratoire de l'Informatique du Parallélisme (LIP)

À Cléa et Teo.

Remerciements

Je commencerai ces remerciements par un reproche à tout ceux qui avant moi se sont retrouvés devant cette tâche. Je leur reproche d'avoir rendu ce moment des remerciements si conventionnel. Je leur en veux par conséquent de réduire la sincérité et la portée de ce que vous aller lire ici. Une Habilitation à Diriger des Recherches se veut être le résultat de travaux d'équipes ou de collaborations et par conséquent ne peut exister sans remerciements. Mais dans le cadre de ces travaux, le travail d'équipe est j'en suis persuadé la véritable contribution. Cela risque de nuire à l'intérêt que vous pourriez porter à ce manuscrit mais... mais, je n'ai rien fait, juste rencontré les bonnes personnes au bon moment.

Parmi elles je commencerai par les membres de mon jury. Ce jury est à mes yeux bien plus important qu'une simple règle administrative à remplir. Ce jury a été composé surtout de personnes que je respecte, que j'admire profondément et qui chacun à sa façon m'ont transmis un peu de leur passion pour la recherche. Merci à vous aussi d'avoir rapporté ce travail, j'espère que vous avez pris autant de plaisir à le découvrir ou le redécouvrir que j'en ai pris à le vivre. C'est la meilleure façon que j'ai trouvé pour vous dire, merci.

Au commencement de cette aventure, une personne, Frédéric Desprez. D'un point de vue professionel je lui dois tout. A l'issue de ma thèse il m'a donné une chose d'une valeur inestimable. Il m'a donné sa confiance et il a cru en moi. Mais comme il ne fait jamais les choses à moitié, ceux qui le connaissent savent de quoi je parle, il ne s'est pas arrêté là et au fil des années m'a donné les moyens de réaliser les projets qui nous tenaient à coeur. Le binôme que nous avons constitué pendant l'intégralité de ces travaux a fonctionné dès les premiers temps et aucun grain de sable n'est jamais venu perturber la qualité de nos rapports. Fred, j'ai toujours eu à coeur de te montrer que tu avais eu raison de me faire confiance, j'espère avoir réussi. Je ne parlerai pas de notre amitié, cela ne concerne que nous, cependant je voudrais profiter de cette espace pour faire taire à jamais une rumeur: «Non, nous ne sommes pas mariés». Merci pour tout Fred, et pour le reste aussi.

Comme je le disais, Fred m'a rapidement impliqué dans le montage de projets divers et variés afin de nous donner les moyens de réaliser nos ambitions. L'objectif premier étant de pouvoir s'adjoindre le concours de collaborateurs de tout horizon. Et c'est là que la magie de DIET a commencé. C'est là que j'ai rencontré nombreux d'entre vous qui doivent être en train de lire ces lignes. Dans une première version de ces remerciements, j'ai trouvé amusant de relater pour chacun d'entre vous une petite anecdote de moments partagés. Et puis après 72 pages je me suis dit que cela n'était pas sérieux. Alors je vais plutôt tenter de factoriser ma gratitude même si pour moi vous avez tous été si différents, si enrichissants individuellement et ensemble. DIET ne peut et ne pourra jamais être associé à un seul nom. DIET c'est vous. C'est vous dans chaque ligne de codes que vous avez écrit, c'est vous à partager vos idées dans nos *working group* hebdomadaires, c'est vous dans chaque ligne de papier à laquelle vous avez participé, c'est vous dans chaque présentation que vous avez donnée, c'est vous à supporter mon humour parfois moqueur, parfois acerbe, parfois méchant, parfois pitoyable, parfois drôle (si, si, c'était un mardi). Vous allez peut être retrouver votre nom dans ces pages, peut être pas mais qu'importe ce que vous avez construit, ce que vous avez fait je

pense que vous pouvez en être fier. En tous cas, j'ai toujours essayé d'être là chaque jour, chaque nuit pour que ce soit le cas. Une grande partie de ma motivation vient de vous. Merci.

Je vais tout de même, sortir quelques noms du chapeau. Le choix était difficile, non, pardon, le choix était impossible. Alors HDR oblige je me suis dit que réduire injustement cette liste à mes doctorants serait permis. Martin tout d'abord, qui a essuyé mes premiers pas dans l'encadrement de doctorant et quelle fierté de le retrouver un jour présentant avec succès les résultats de son projet ANR, que de chemins parcourus mais chemins qui se croisent souvent pour mon plus grand plaisir. Pushpinder ... non Push, je sais que tu n'aimes pas ce surnom et comme peu doivent l'employer en Irlande, il garde donc la trace unique de nos heures passées ensemble. Et s'il y a eu des bas ça a toujours été pour aller plus haut. Tu m'as beaucoup appris. Vincent, co-encadrement à distance, pas toujours facile de tisser des liens, et pourtant je ne sais pas si c'est la chaleur des calanques marseillaises ou simplement toi mais tu as réussi. Cédric, au delà du rapport encadrant/thésard l'alchimie qui a eu lieu dans ces années de collaboration est à jamais en moi, et si tu es encore à mes côtés régulièrement à différents niveaux personnel ou professionnel ce n'est pas le fruit du hasard. Tu es vraiment quelqu'un d'exceptionnel, alors surtout n'oublie pas le sage conseil que l'on t'a prodigué "Attention à ne pas te Caroniser". Benjamin, autre histoire, autre symbiose. Au moment ou j'attaquerai le premier slide de ma soutenance d'HDR, tu seras docteur. J'ai honte tellement j'ai abusé de la confiance que j'avais en toi, tellement abusé de ta fiabilité et de tes si nombreuses qualités professionnelles ou humaines. Bras droit infaillible aux yeux de tous, nous savons tous les deux que cela va beaucoup, beaucoup plus loin. Il est peut être trop tôt pour ajouter mes remerciements à Adrian, alors simplement une recommandation. Adrian merci de ne pas prendre contact avec tes prédécesseurs, juge par toi même, cela me laissera une petite chance. Par ces quelques lignes j'espère vous remercier et remercier tous les stagiaires, ingénieurs qui du coup comprennent qu'il m'est impossible de parler de vous tous. Merci, et si j'obtiens mon HDR, je vais militer pour que les thèses durent dix ans, car trois ans avec vous c'est bien trop court.

Il est impossible que je m'attaque à cet exercice des remerciements sans avoir quelques mots pour vous. Anne-Pascale, Caro, Coco, Evelyne, Isa, Marie, Sylvie, ... J'aurais bien placé vos noms par ordre de préférence si j'en étais capable histoire de provoquer une séance coin café animée comme nous en avons partagé tant mais j'ai opté pour un ordre alphabétique. Je pourrais souligner ici la qualité et l'importance de votre travail de collaboratrices à bien des projets. Mais je l'ai déjà fait tant de fois, et je mets toujours tant d'énergie à transmettre et partager la reconnaissance que l'on vous doit. Alors non je ne vous remercierai pas pour cela, pas aujourd'hui. Aujourd'hui je voudrais juste vous remercier pour tout ce qui ne peut pas s'écrire ici, pour tout ce qui ne concerne pas nos relations de travail mais qui a existé. J'espère avoir encore longtemps la chance, tradition oblige, de vous envoyer des clins d'oeil "*Starbuckiens*" de l'autre bout du monde, et de vous faire rire avec mes frasques et péripéties réalisées en collaboration avec Air France et la SNCF. Et vous savez que nous n'avons pas ménagé nos efforts, pour votre plus grand plaisir. Merci, ne changez pas, jamais.

Il y a encore tant de noms de collègues qui ont croisé ce parcours et que j'aimerais citer ici. C'est terriblement frustrant. Tant d'histoires partagées qu'elles soient professionnelles ou personnelles, tant de tableaux blancs gribouillés. Alors je vais être à nouveau injuste, mais symboliquement à travers lui pour des raisons similaires ou d'autres, j'espère que vous vous sentirez remercié. Je voulais adresser un merci tout particulier à Yves Robert. Pour un conseil lancé sur un problème de

recherche, un autre conseil lancé sur un dossier à monter, encore un autre conseil pour gérer mes enseignements puis cet autre conseil juste pour aider à ce que la journée soit moins difficile. Et merci pour ton humour, cela m'a aidé dans mes relations avec mes autres collègues qui pouvaient toujours se dire, ça va.. y'a pire. Mais surtout merci, car nous savons tous les deux qu'au delà des apparences les personnes sur qui l'on peut réellement compter quelles que soient les circonstances sont infiniment faibles. Je crois que ça s'appelle un ami.

Et je terminerai par seulement quelques mots en toute pudeur pour ma petite famille. Je ne veux pas non plus vous dévoiler ici, mais tout cela n'a un sens que parce que vous existez. Teo, Cléa vous ne saurez jamais ce que vous avez inscrit en moi et ce que vous représentez à mes yeux. J'espère que j'en serai digne, et j'espère qu'un jour vous aurez le plaisir de faire un travail qui vous offre autant de plaisir, et que vous y puiserez le goût de l'effort et du défi. Vous savez... "*si c'est pas dur, c'est pas drôle*". Merci à toi, Isabelle, tu sais mieux que quiconque que j'ai des petits conflits avec les limites mais tu as toujours su me ramener les pieds sur terre. Merci pour tout ce que tu fais pour moi, pour tout ce que tu fais pour nous.

Enfin, merci de votre fidélité à [vag] et [bae]. Merci de faire parti de tout cela.

A bientôt... Un petit Starbucks ?

Contents

Chapter 1

Introduction

10 years. Ten years of research around high-performance computing in a distributed environment. And throughout these years, the development of the DIET middleware, increasing the value of that research. Today, a start-up company [1] improves the status of DIET, giving it a new life. It feels only natural for me to give a complete review of this decade.

The purpose of this document is to give a comprehensive description of the DIET middleware. Researches and developments carried out to create this Grid and Cloud middleware will be discussed. Most of my researches focuses on DIET, hence the target of this document is twofold. First, to give an overview of my work and, secondly, to illustrate, through a real middleware, the issues met when designing, deploying and using this software platform dedicated to high-performance computing.

Huge problems can now be computed over the Internet thanks to Grid Computing Environments or the new approach called Cloud Computing.

Because many of the current applications are numerical, the use of high-performance libraries is mandatory. The integration of such libraries to high level applications using languages like Fortran or C is far from easy. We increased the capability of DIET and generalized the services access from high-performance library to many applications with the same kind of requirement an important need of a large set of resources. Moreover the needs in computational power and memory of such applications may not be available on every workstation. For these reasons, the RPC paradigm seems to be a good candidate to build Problem Solving Environments (PSE) for numerical applications on the Grid. Several tools going with this approach exist, like NetSolve [20] or Ninf [131]. They are commonly called Network Enabled Server (NES) environments [124].

1.1 The DIET environment platform: From cluster to the Cloud through the Grid.

The genesis of DIET was introduced in the HDR of Frédéric Desprez [74] as the next step to provide a distributed architecture for numerical libraries, and the way was to develop a parallel version of Scilab [42]. This architecture shown in my Ph.D introduces the hierarchical point of view. During the meeting around this architecture, the problem of scalability was introduced and

[1] http://www.sysfera.com

Frédéric Desprez drew a picture of a hierarchical and distributed architecture (Figure **??**), DIET was born.

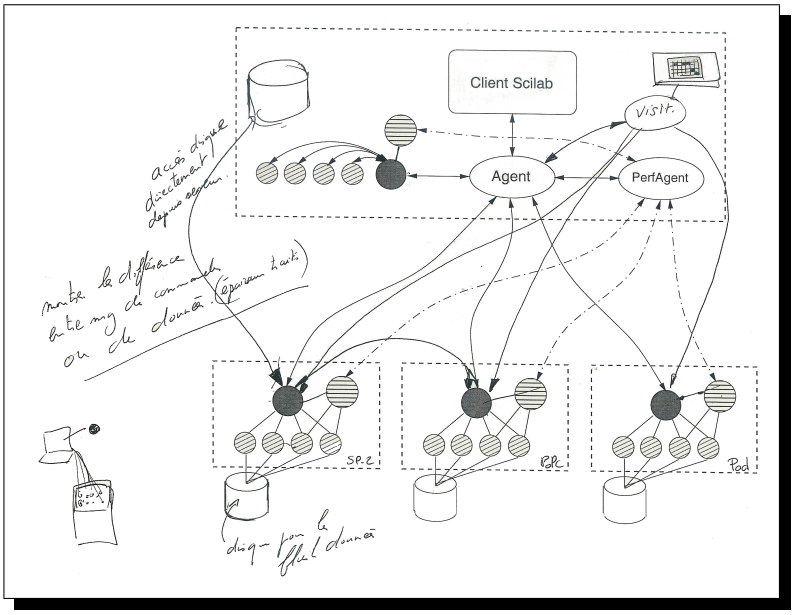


Figure 1.1: Based on a proposal for the Scilab$_{//}$ architecture, DIET was born.

Dealing with high-performance computing, we needed the maximum amount of resources. Thus, we worked to provide a transparent access to heterogeneous resources.

1.1.1 A view from the Cluster

During my Ph.D, the parallelism on SMP was famous. Taking advantage of the progress in networks and the cost of workstations, cluster of computers in Universities and companies were the latest fashion. Many applications benefited from clusters through the parallel version of the application, using a message-passing library such as MPI. Thanks to the success of this architecture, the need to manage this kind of resources became critical. There are two kinds of use cases:

- Shared method: in this case, some instances of the same application or from different applications can be running simultaneously. The first versions of DIET targeted this kind of architecture and task scheduling took into account the shared resources between each users.
- Dedicated method: the problem of the previous method is the lack of quality of service. Thus, it is impossible to guarantee an execution without disruption. This disruption can decrease performances. LRSM (Local Resources Management System) are designed to solve this problem and manage clusters architecture. OAR [40], SGE [86], LSF [102], Loadleveler [139], etc., give mechanisms to reserve a set of resources and ensure a unique access to them.

DIET complies with these two methods. In the first case, the resources are accessed directly through DIET, which is deployed before the beginning. This method can be done using LRSM too, but the reservation must occur before the deployment. The resources reservation mechanism

is thus outside DIET and unmanageable from DIET point of view. In the second case, as we will see in Chapter 4, DIET is able to transparently manage the access to a non-finite set of LRSM.

1.1.2 A view from the Grid

Thanks to the success of cluster, the Grid concept was introduced. The idea was to give a transparent access to distributed and heterogeneous computing resources.The word Grid is imputed to Ian Foster [83] and is used by analogy with the US power infrastructure. Infrastructures have been created to interconnect computers and/or clusters in a WAN environments. Projects were created such as the EGEE Grid [119] to manage the data provided from the Large Hadron Collider (LHC), or French Grid project called Grid'5000 [31], and more recently the US version with the Future Grid project [2]. One way to create a Grid is to aggregate clusters together. This architecture is based on Cluster of Clusters through a high speed network. For instance, in Grid'5000 the network is configured to allow direct connections between different sites, avoiding the problem of firewalls. DIET was designed to deal with these platforms, with the purpose of cutting off grid heterogeneity. DIET offers the user a simple API to manage data distribution and computing in the Grid.

1.1.3 A view from the Cloud

Since 2006 the concept of distributed computing has a new face, or at least a new name. Cloud computing now has important media coverage, and large companies such as Amazon, Google, IBM and Microsoft provide an access to their systems. Different clouds exist:

- Public Clouds, where the platform is available from an external access through a web application or a web service.
- Community Clouds, closed to the public Cloud, but dedicated to a community of users sharing the same needs.
- Private Clouds, where the Cloud platform is deployed on a private network. For security reasons, a lot of companies prefer to deploy their own Cloud platform.

Following the same way as Cluster and Grid, the Galaxy (or Intercloud) computing computing is born. We can imagine to work on a collection of Clouds (Cloud of Clouds). Many companies believe in a new stage for informatics and computing. As Richard Stallman, founder of the Free Software Foundation, said: "The interesting thing about Cloud computing is that we've redefined Cloud computing to include everything that we already do" [3]. Nevertheless, if it is not actually new,it offers large platforms to exploit distributed and grid-specific algorithms. The amount of resources is so large that we talk about elastic computing. For a middleware, that means providing a high scalability.

As we will see in Chapter 4 we have extended DIET's capabilities from managing one or many clusters or dealing with a Cloud computing platform. Resource management is similar, but virtual machine management must be added. The notion of service with admission is easy to take into account using an appropriate scheduler. A cost parameter can easily be integrated into DIET as a

[2] http://futuregrid.org

[3] `http://www.guardian.co.uk/technology/2008/sep/29/cloud.computing.richard.stallman`

scheduling parameter. Regarding deployment, mapping algorithms can be more dynamic to manage elastic computing [85, 128] and update resources reservation according to platform load.

Thus, the context of my research was to contribute to help application users to use the architectures we have seen during this introduction. The point is to hide platform heterogeneity with performance optimisation as an objective function, and to minimize the amount of resources used. This can be seen as the "cost-performance" ratio. The benefit of driving research about a middleware in development is that, since it implies providing an end-to-end solution, a wide scope of research topics must be studied.

1.2 DIET research topics

Through the DIET middleware, we are dealing with the whole issue of the GridRPC environment. These kinds of environments require **Interoperability** capabilities to interact together. We need to have an appropriate problem description. Moreover, beyond this word, what we mean is the capability for different middleware to communicate together and exchange information or computing requests. One way to achieve that is to design a standardized API, as GridRPC does. Chapter 2 Section 2.2.1 gives an overview of the GridRPC paradigm. This chapter also introduces the DIET architecture. Given theses bases, we can address the related research topics we have worked on. In this chapter, we also propose a solution to the **resource localization** problem. Indeed, the point of a middleware is to make the bridge between users and resources. The middleware should know where the hardware resources are as well as the software one. It is important to localize the resources and their availability. Thus, static and dynamic information should be taken into account. Moreover, service descriptions using ontology methods can be useful.

Scalability is dealt with Chapter 3 through a distributed servers/agent(s) platform, the aim is to provide access to a large amount of servers, for a large amount of clients. DIET has been designed around this kind of hierarchy but we will see that a crucial way to performance is to provide a good deployment of these elements. We can notice that scalability is a real challenge when dealing with the visualization. Distributed middleware is complicate to visualize. Visualizing it becomes a nightmare when the system grows.

Behind the scalability problem we can discuss about **Service discovery** problems. As we mentioned before, the resources access can be classified into two families: hardware and software. The first one can be associated to resources as cluster or computing nodes, the second one can be associated to service access. How can we reach the right service (the right application)? We will see in the Chapter 3 Section 3.4 a general solution to provide a scalable service discovery system beyond the scope of DIET.

After addressing few problems as a clients' point of view, we focus on the problem of **Resource management** in Chapter 4. As we mentioned, DIET is a middleware designed to associate computing request to available resources. These resources can be known at the deployment step, or the resources can be reached and reserved during runtime. In Grid and Cloud environments the resources can be accessed after a reservation through a LRMS (Local Resources Management System) or Batch Systems such as SGE [86], OAR [40], LSF [102], LoadLeveler [139], etc. DIET can submit the appropriate script to the LRMS and reserve the amount of resources required by the DIET scheduler.

The next Chapter deals with the needs to deploy the whole software infrastructure described in the previous chapters, and how to do it. The notion of deployment in the case of distributed

middleware consists in the deployment of software components of this middleware. The aim of this deployment is to know where each part of the middleware is launched; effectively, it is “planning". To obtain good results, two parts are closely linked: the design of the architecture (the tree in the case of DIET), and which resources are associated to each software component. Chapter 5, resulting from two Ph.D theses, focuses on the issues of **deployment and planning**. In the case of dynamic platforms, the available resources are subject to changes. Thus, we need to provide the correct resource localization. The deployment of agents and servers should be adapted to the new configuration. We can notice that **Dynamic platform** should be taken into account at the deployment level and at the scheduling level. We are now ready to study the runtime part. The main problem to address is **Data Management** (Chapter 6). Many applications require a large amount of computing manage a large amount of data. The concept of Data Grid [61] is a kind of dedicated Grid meant to deal with this problem. According to the GridRPC paradigm, the data-persistence mechanism can help avoid useless data transfer. Moreover, data (re)distribution between servers or set of servers is a complex research field, on which we have been working during the ARC RedGRID [4]. Garbage collection is another problem that we need to enquire.

The second problem to address is a large and complex topic, **Scheduling**. As we will see in Chapter 8, scheduling of computing tasks on a distributed and heterogeneous platform is a real challenge. DIET was created to offer a platform to design and validate such as schedulers. In too many cases this problem is NP-hard, thus, heuristics must be implemented. DIET offers plug-in scheduler abilities, this means that a developer who knows the application can create his own scheduler. Distributed schedulers are a good way towards scalability but they add complexity due to the distribution of the scheduling knowledge. DIET must deal without a centralized scheduler. The “simplest" way is to consider on-line scheduling, but the quality of this approach depends of the task's arrival time. To increase performance, we can imagine buffered requests, and works on semi-static scheduling or even on off-line scheduling. A research topic intrinsic to scheduling is the **performance evaluation** problem. A scheduler needs to have various pieces of information to be efficient. In order to make the right decision and, find the best resources, we need to know the current status of available resources (CPU load, memory load, bandwidth, network latency, I/O performance, etc.). Likewise, to find the best task mapping, we need to have an estimation on the time taken by each task. The forecasting of the execution time is a key point in many research scheduling problems, but very often difficult to obtain. Indeed, the amount of communications, the CPU usage and the amount of disk access are difficult to predict, and it is very difficult to know the corresponding execution time.

Chapter 7 gives DIET an extension to address the **workflow management** topic. For a family of applications, the global execution is a set of tasks. These tasks have some dependencies between them. A given task cannot start until the task it depends on is finished. In many applications, the dependency between tasks is due to data exchanges between them. DIET has introduced a workflow engine to deal with this kind of applications. In the case of DIET we can consider that a complete workflow one request, thus DIET is used as much to schedule one workflow as to schedule different independent workflows in parallel.

To be exhaustive around the research topic, we must talk about another related problem: **security**, which is a very important point, in particular for private companies. Authentication and authorization must be taken into account. The difficulty is to have a valid certificate authority. In the distributed DIET architecture, for scalability reason, authentication should be distributed

[4] `http://graal.ens-lyon.fr/~desprez/REDGRID/index.html`

too, to avoid a new bottleneck. Another point about the security is the need for data transfer. Encryption could be required by users. This problem is out of scope of this document. However, development conducted by SysFera start'up led to introduce security within DIET.

Finally, in Chapter 9 we show how DIET was used to serve different kinds of real applications—cosmological, robotic, etc. We present in particular the example of the Decrypthon project, which uses DIET at a production level.

Chapter 2

DIET Architecture

Contents

2.1 Introduction

In this chapter we introduce an overview of the DIET architecture. As we have seen during the introduction, a GridRPC environment usually have five different components: (1) Clients that submit problems they have to solve to (2) Servers, (3) a Database that contains information about software and hardware resources, (4) a Scheduler that chooses an appropriate server depending on the problem sent and the information contained in the database, and finally (5) Monitors that acquire information about the status of the computational resources. In the following we introduce each components.

2.2 Clients

From the Grid middleware point of view, the client is the interface between the user and the Grid computing. Through this client, the user is able to submit her computation. Grid middleware are designed to give a transparent access to resources. This access should be as easy as possible. The request submission must be straightforward. Nevertheless, to be efficient the Grid middleware requires knowledge from the client. We need to reach a compromise to deal between simplicity and middleware requirements. Three level can be considered when talking about the client:

User level: The client of the platform is the final user. No specific application knowledge is required. Only the input parameters are needed. The user's access to the platform is given through a high level client such as a web page, a numerical computational software like Scilab, or a program (binary, java program or even a Python script).

Application client level: At this level, the user needs to know her application, and have a thorough knowledge of it to perform its integration with the middleware. At this level the user gives the information to launch the computing service, the data required for the computing, runtime information (i.e., as in the case of workflows that will be discussed in the Chapter 7). In order, to ease integration with the middleware, DIET chooses to provide an API as simple as possible. Two API exist for DIET: an API reducing the functionalities but compliant with the GridRPC, and an API designed for DIET to access all available features. Using a few function calls the user can create a client for the user at the previous level.

DIET client: The named client is hidden from the user's point of view, but is dedicated to the diet developer. It corresponds to internal software architecture that deals with the API and manages the communications between users and schedulers (Section 2.3).

A **client** that has a problem to solve should be able to obtain a reference to the server that is best suited for her. A problem can be submitted from a web page, a PSE such as Scilab (a Matlab-like tool), or from a compiled program. DIET is designed to take into account the data location when scheduling jobs. Data are kept as long as possible on (or near to) the computational servers in order to minimize transfer times.

2.2.1 GridRPC API

Joint work with:

⋆ Yves Caniou	: University Claude Bernard 1. LIP Laboratory. Lyon. France.
⋆ Frédéric Desprez	: INRIA. LIP Laboratory. Lyon. France.
⋆ Hidemoto Nakada	: National Institute of Advanced Science and Technology. Tokyo. Japan.
⋆ Yoshio Tanaka	: National Institute of Advanced Science and Technology. Tokyo. Japan.
⋆ Keith Seymour	: University of Tennessee. Knoxville, TN. USA

For the client, the usage of DIET corresponds to writing a program with different function calls. DIET is compliant with the GridRPC standard, and we are involved in the standardization process. One of the goals of the GridRPC API is to clearly define the syntax and semantics for GridRPC, which is the extension of the Remote Procedure Call (RPC) to Grid environments. Hence, end-user's client/server applications can be written given the programming model.

The GridRPC Paradigm

The GridRPC model is pictured in Figure 2.1: (1) servers register their services to a registry; (2) when a client needs the execution of a service, it contacts the registry and (3) the registry returns a handle to the client; (4) then, the client uses the handle to invoke the service on the server and (5) eventually receives back the results.

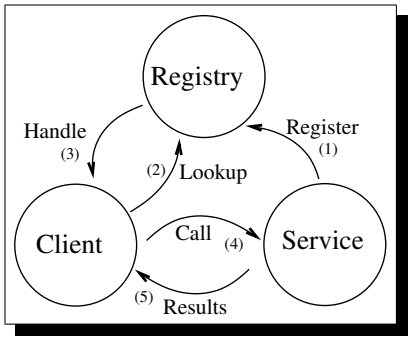


Figure 2.1: The GridRPC model.

The GridRPC API

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

GridRPC Data Types

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

GridRPC Functions

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

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

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

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

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

Presentation of the Interoperability Between Implementations

The Open Grid Forum standard describing the GridRPC API did not focus on the implementation of the API. Then, divergences in implementations have been observed. In order to make a GridRPC client-server code reusable in all GridRPC middleware relying on the GridRPC API, a document listing requirement for interoperability between implementations has been proposed [159] points out, with exhaustive test-cases, the differences and convergences in behavior of the main GridRPC middleware implementations, namely DIET, Netsolve, and Ninf. In addition, a program has been written to test the GridRPC compliance of all middleware.

2.3 Schedulers

The current environments previously cited have a centralized scheduler which can become a bottleneck when many clients try to access several servers. Figure 2.2 shows two experiments performed with DIET with the scheduler either in a distributed arrangement or in a centralized arrangement to manage 150 nodes.

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. Note that the DIET scheduler is designed to allow distributed configurations, and is therefore not perfectly optimized for a centralized configuration. Although it would be interesting to compare against a centralized-only scheduler as well, we feel that these results are compelling. At least for the DIET toolkit, and most likely for other schedulers as well, distributing the task of scheduling can improve performance in large resource environments.

In the case of DIET, the scheduler is scattered across a hierarchy of **Local Agents (LA)** and **Master Agents (MA)**. An MA receives computation requests from clients. These requests are generic descriptions of problems to be solved. From the MA, requests reach servers through the hierarchy of LA.

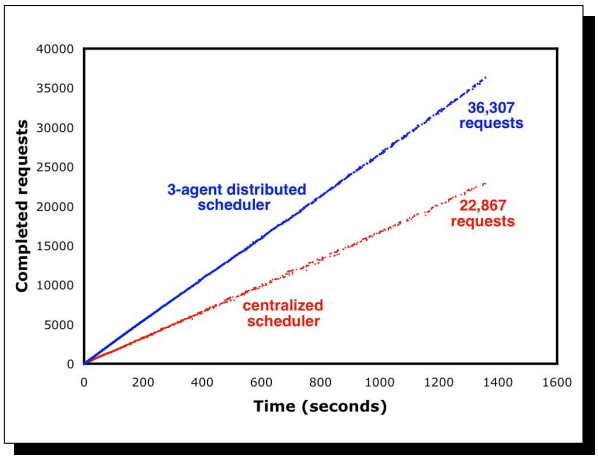


Figure 2.2: Comparison of the numbers requests completed by a centralized DIET scheduler versus a three agent distributed DIET scheduler.

2.4 Servers

DIET is built upon **Server Daemons (SeD)**. A SeD encapsulates a computational server. The information stored on an SeD is the list of problems that can be solved on it, and every information concerning its load (available memory, number of available resources, ...). A SeD declares the problems it can solve to DIET and provides an interface to clients for submitting their requests. A SeD can give performance prediction for a given problem.

An MA collects information about computation services from the SeDs and chooses the best one. The reference of this server is returned to the client. An LA aims at transmitting requests and information between MAs and SeDs. The information stored on an LA is the list of requests and, for each of its sub-trees, the number of servers that can solve a given problem and information about data distributed in this sub-tree. Depending on the underlying network architecture, a hierarchy of LAs may be deployed between an MA and its SeDs.

2.5 Communication Layer

Our first DIET prototype is based upon OmniORB, a free Corba implementation which provides good communication performance. 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 [72]. Moreover, the time to select a server using Corba should be short with regard to the computation time.

Moreover Grid computing is a very active research domain. Existing platforms are usually subject to frequent experimental modifications and feature add-ons. Object oriented development platforms allow an easier development and a greater maintainability of the code. Corba is thus well suited to support distributed resources and applications in a large scale Grid environment. New dedicated services can be easily published, as well as existing services can be easily used. Thus we can conclude that Corba systems are one of the alternatives of choice for the development of Grid specific services. Another alternative to OGSA the most famous way to develop Grid

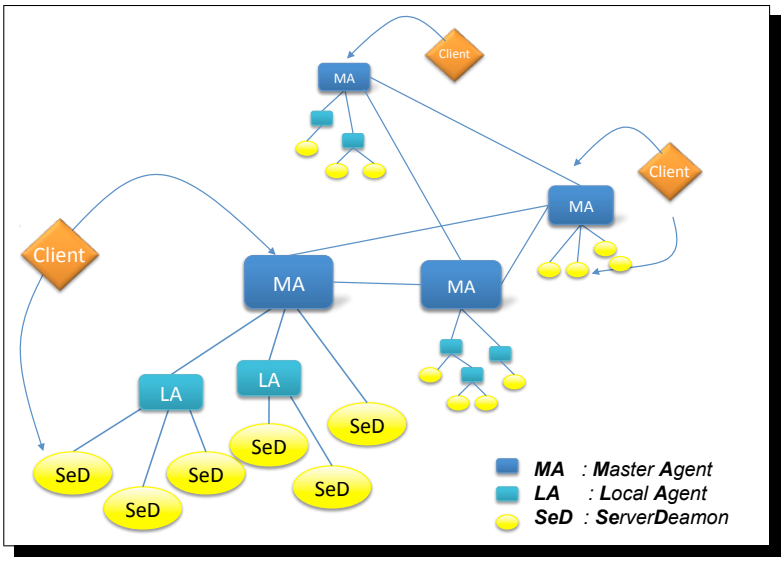


Figure 2.3: DIET architecture.

services [79, 84].

2.6 DIET in Practice

The DIET architecture is built in the hierarchical order, each component contacting its father. The MA is the first entity to be started. It waits for connections from LAs or requests from clients. When an LA is launched, it declares itself to the MA. At this point of the system initialization, two kinds of components can connect to the LA: a SeD, which manages some computation resource, or another LA, to add a hierarchical level in this branch. A client can contact an MA to determine the best server, and then, directly deal with it.

The following algorithm allows an MA to choose a server to execute a computation. This decision is taken in four steps. (1), find the problem; (2), locate involved data and capable servers by sending a request to computational servers, propagate a request to its sub-trees; (3), forecast the computation time on all capable servers, and send the reply to the request back to the MA; (4), choose a server and send its reference to the client. This is done by the MA once it has collected all replies.

The algorithm used to reply to the request forwarded by the MA is divided in three steps. Step 1, **Initialization** when a SeD receives a request, it sends a response structure to its father. If the server can solve the problem, it keeps the evaluated computation time acquired from a forecast tool. Step 2, **Aggregation**, each LA gathers responses coming from its children and aggregates them into one structure. The information concerning communication times are gradually upgraded. The forecast tool computes the transfer time of data to the capable servers, combining information from monitoring and data attributes. This transfer will use the shortest path among those that are monitored. Step 3, **Use**, when the responses come back to the MA, it can use them to take a decision. The evaluated computation and communication times are used to find the server with the lowest response time to perform the computation.

2.7 Using DIET: a User Point of View

When this architecture has been introduced, recurrent questions kept on coming from users. How can they interface their applications with DIET? Where and how they had to contribute? Figure 2.4 gives the main idea of that. Following the picture with a up-bottom view we described this layered structure.

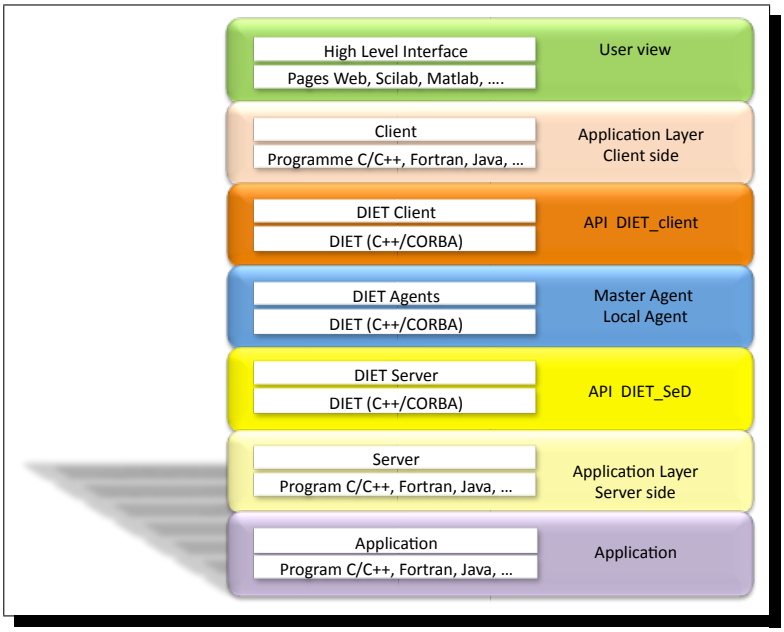


Figure 2.4: DIET layers. User point of view.

At the top level (User view) a high level interface can be designed (not required but more user-friendly) such as web pages (e.g., the web pages created for the Décrypthon project [22], or for the TLSE project [50], or the web portal designed for the ANR project LEGO [1] for cosmological application [39]), or through high level tools such as Scilab (i.e., an interface between DIET and Scilab was designed, and Scilab is able to send requests to DIET). Through these interfaces we reach the maximal isolation of the platform. The user gives parameters without having to deal with Grid or Cloud resources.

The second layer (Application Layer - Client side) corresponds to the first contribution of the user to integrate her application in a Grid environment and more precisely to DIET in our case. The idea was to provide an API as light as possible (keeping the maximum of features). As already said, the API of DIET is twofold. A reduced API, but compliant with the GridRPC standard [152], and a dedicated more extensive API. For the sake of simplicity, we can see these API as the functions required to submit to the middleware the data needed for the computation. The call to DIET is performed from this API (i.e., respectively `diet_call()` and `grpc_call()`)

The layers API DIET _client, Master Agent/Local Agent and API DIET _SeD correspond to the DIET core and thus the user does not have to deal with this layer. Moreover, we can notice

[1] `http://graal.ens-lyon/LEGO`

that Corba is used only in these layers, this implies that the user does not need any knowledge of this paradigm.

The next layer is the dual part of the Application Layer - Client side but on the server side. The aim here is to grab the data from the middleware and launch the application with these data. This step requires to have knowledge on how to launch the application. Indeed, this layer will choose if the execution will be launches in sequential or in parallel mode.

Finally, the last layer is the application. In many cases DIET does not know anythings about the core of the application. It could be useful (but not necessary) to have more knowledge about it, as the expert of scheduling wants to take benefit of this information to design a better scheduler. Nevertheless, DIET does not require to update an existing application to be integrated with it. This point of view is very appreciated from users. To be more accurate, DIET implies only one constraint which is that the application must be called through a line command or through an API, or even through a toolkit (e.g., gSOAP [167] to deal with the Web Services).

2.8 DIET Suite

To help the DIET user community, we have developed a set of tools to ease the usage of this middleware, the deployment of all the distributed software components, and the visualization of each component. The implementation of these external tools was driven accordingly to the user's requirements. Figure 2.5 shows how these external tools work together. We give here a brief introduction to each of them. The details of the implementation of each tool is out of topic of this manuscript.

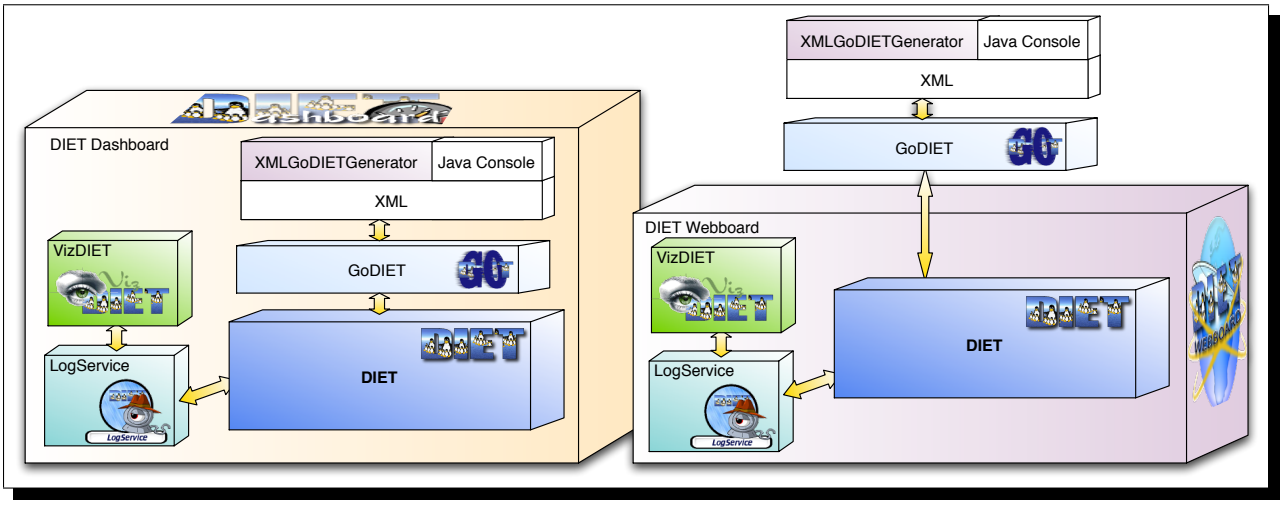


Figure 2.5: DIET and tools.

2.8.1 GoDiet

Joint work with:

⋆ Holly Dail : INRIA. LIP Laboratory. Lyon. France.

GoDiet 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; we have chosen Java for its implementation as it satisfies all of these requirements and provides rapid prototyping.

GoDiet automatically generates configuration files for each Diet component, taking into account the user's configuration preferences, and the hierarchy defined by the user. It also 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.

GoDiet requires an XML file as input, in which the user describes her available compute and storage resources, and the desired overlay of agents and servers onto those resources. In short, the GoDiet XML file contains the description of Diet agents and servers, and their hierarchy, the description of desired complementary services like LogService, the physical machines to be used, the storage disks 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 resources description and the deployment configuration description; the resources description portion must be written once for each new Grid environment, but can then be re-used for a variety of deployment configurations. Experiments to validate GoDiet have been presented in [43].

2.8.2 XMLGoDIETGenerator

Joint work with:

⋆ David Loureiro : INRIA. LIP Laboratory. Lyon. France.

XMLGoDIETGenerator is a Java application that builds, given a compact descriptor file and a reservation directory a GoDiet XML file you can use to deploy a Diet platform with GoDiet.

XMLGoDIETGenerator relies on a small set of information in order to be as simple as possible to use. Moreover the information are close to the ones used in the GoDiet XML file in order to be understandable for the developers that write their one GoDiet XML files.

This application fills the lack of an XML Generator for large experiences that need large XML files for GoDiet, a lack of a generator that could be flexible by offering the users a certain number of predefined hierarchies if none were satisfying. There is also the possibility of creating your own hierarchy by implementing your own Hierarchy class.

Finally, one can consider the fact that for large experiments, writing the GoDiet file by hand is time consuming. And if the user should redo this experiment with a different set of machines, or cluster, the modifications that need to be done on the GoDiet file can be reduced with XMLGoDIETGenerator, as the file will be generated according to the available resources.

XMLGoDIETGenerator is a stand-alone application you can use from the command line for an easy creation of an XML GoDiet input file. But it can also be used from the Diet Dashboard, through a nice GUI interface helping the user in creating the GoDiet file with or without keeping

the input file of the XMLGoDIETGenerator.

2.8.3 LogService

Joint work with:
- ⋆ George Hoesch : TU München. Germany.
- ⋆ Cyrille Pontvieux : IUP informatique. Besançon. France.

An event monitoring system called LogService [32] has been designed. This monitoring service offers the capability to be aware of information that need to be gathered from a distributed platform. The communication layer of LogService is based on Corba technology. A LogComponent is attached to each component that needs to to spawn information events. A LogComponent relays information and messages to LogCentral. LogCentral collects messages received from the various LogComponents, then it stores or sends these messages to LogTools. LogTools connect themselves to LogCentral and wait for messages. The visualization requires to centralize the information. The main interest in LogService is that information are collected by a central point LogCentral that receives logEvents from LogComponent that are attached to the component that you want to monitor. The LogCentral offers the possibility to forward this information to several tools (LogTools) which are responsible for analyzing these messages and offering a comprehensive information to the user.

LogService defines and implements several functionalities.

Filtering mechanisms are used to reduce the number of messages sent. In order to decide which messages are required by a tool. The tools have to declare their filters to the monitor (LogCentral).

Event ordering is another important feature of a monitoring system. LogService handles this problem by the introduction of a global time line. When created, each message receives a time-stamp. The problem that can occur is that the system time can be different on each host. LogService measures this difference internally and corrects the time-stamps of incoming messages accordingly. The time difference is corrected using the time-stamp of the last ping that LogCentral sent to the LogComponent. However, incoming messages are still unsorted. Thus, the messages are buffered for a short period of time in order to deliver a sorted stream of messages to the tools. Messages that arrive out of order within this time frame are sorted in the buffer and can be properly delivered. Although this induces a delivery-delay for messages, this mechanism guarantees the proper ordering of messages. As tools usually do not rely on true real-time delivery of messages this short delay is acceptable.

Dynamic system state: Components may connect and disconnect at runtime. A problem that arises in distributed environments is to know the state of the application at a given time. This state may for example contain information on connected servers, their relationships, the active tasks, and many other pieces of information that depend on the application. The system state can be constructed from all events that occurred in the application. Some tools rely on this state to work properly. The problem appears if those specific tools do not receive all messages. This might occur as tools can connect to the monitor after the application has been started. In fact, this is quite probable as the lifetime of the distributed application can be much longer than the lifetime of a tool. As a consequence, the system state must be

maintained and stored. In order to maintain a system state in a general way, LogService does not store the system state itself, but all messages which are required to construct it. These messages are identified by their tag and stored in a special list. This list is forwarded to each connected tool. This process is transparent for the tool since it simply receives a number of messages that represent the state of the application. In order to further refine this concept, the list of important messages is also cleaned up by LogService. After a disconnection of a component the respective information is no longer relevant for the system state. Therefore, all messages sent by this component is removed from the list.

2.8.4 VizDIET

Joint work with:

⋆ Raphaël Bolze : CNRS. LIP Laboratory. Lyon. France.

The first goal of VizDiet is to graphically represent the Diet hierarchy and to monitor its behavior. VizDiet gives the possibility to show a lot of information extracted from information events received from LogCentral. All objects and information about the Diet components are used to compute the following properties of the system.

Average time : represents the elapsed time mean for each request.

Max/min time : max/min time of all requests' elapsed time.

Load : the number of requests computed at the same time. It is the number of requests that have a common intersection in the interval time represented by begin and end solve time.

Number of requests : this information is very useful. For example, one may be interested in the number of requests for a specific service on a specific SeD.

Latency : This value represents the Diet's latency that includes the time to transmit data from client to server, network latency, and any other latency introduced by scheduling policy (e.g., request queueing ...).

Scheduling information : Diet's agent return the sorted list of SeD that can compute the service asked by the client.

data information : with the aggregation of data information represented by logEvent, we are able to know the amount of data presents in Diet, but also the time needed to transfer data and history of transfer for this.

Interaction with other systems : As LogService can relay any event, we can monitor the interactions of Diet components with JuxMem [2] [16] and know the amount of data read from and written by JuxMem.

VizDiet can display a variety of information about the activity of a Diet platform. Figure 2.6 show some example of VizDiet statistic output as the load of the chosen element, the flow of requests (very useful for observing the behavior of the scheduler and has been proven very useful for scheduler developers) and the tasks repartition given with the Gantt chart.

[2] JuxMem (Juxtaposed Memory) is a data sharing service for Grid computing

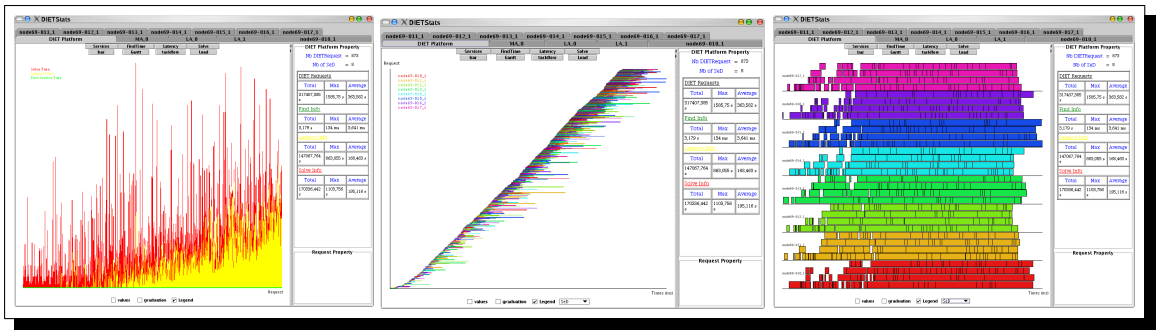


Figure 2.6: Load, taskflow, and Gantt chart in VizDiet stats.

All of these methods can be applied to calculate values for one element such as a SeD, an Agent, or it can be applied to the entire Diet platform. It can also be applied to a specific set of requests restricted to one type of request.

2.8.5 Diet Dashboard

Joint work with:
- ⋆ Abdelkader Amar : ENS-Lyon. LIP Laboratory. Lyon. France.
- ⋆ David Loureiro : INRIA. LIP Laboratory. Lyon. France.

Diet Dashboard is a set of tools written in Java that provide to Diet end-user a friendly-user interface to design, deploy and monitor the execution of applications. Diet Dashboard is an extensible set of graphical tools for the Diet community. With this tool, a non-expert user can manage Grid resources, monitor the Grid itself and manage the Grid middleware by designing its Grid applications or using workflows and then deploying these Grid applications over the Grid platform. The Diet Dashboard offers a large number of modules, created to answer the different needs of tools appearing in a Grid context. The software architecture design of Diet Dashboard makes its extensible (modules can easily be added to the core of the application). Diet Dashboard is currently based on seven tools:

1. **Workflow designer:** This tool dedicated to workflow applications written in Diet provide to the user an easy way to design and execute workflows with Diet. After connecting to a deployed Diet platform, the user can compose the different available services and link them by drag'n'drop. Once the workflow designed, the user can set its parameters and then execute it. Diet Dashboard includes a generic client that can execute any workflow. The results of the workflow and the execution log are displayed.

2. **Workflow LogService**: This tool can be used to monitor workflows execution by displaying the DAG nodes of each workflow and their states. Three states are available: waiting, running and done. The workflow log service can be used in two mode:

   - local mode: can be used if your Diet Grid application can access to your host.
   - remote mode: this mode is useful if your platform is behind a firewall and allow only ssh connections.

3. **DIET Designer**: Allows to the user to design graphically a DIET hierarchy. Only the application characteristics are defined (agent type: MA or LA, and SeD parameters). The designed hierarchy can be stored to be used with the DIET mapping tool.

4. **DIET mapping tool**: Allows the user to map the allocated Grid'5000 resources to a DIET application. The mapping is done in an interactive way by selecting the site, then DIET agents or SeD. For each Grid'5000 site, the nodes (or hosts) are used in a homogeneous manner but the user can select a particular host if needed.

5. **DIET deployment tool**: This tool is a graphical interface to GoDIET. It provides the basic GoDIET operations: open, launch, stop and also a monitoring mechanism to check if DIET elements are still alive (three states are available: unknown, dead and running).

6. **DIET resource tool**: This tool was designed to manage the user Grid resources which is an important aspect of Grid computing. Currently this tool is used only for Grid'5000 platform and provides several operations to facilitate the access to this platform. The main features are:

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

7. **XMLGoDietGenerator**: The DIET Dashboard can call the external tool XMLGoDietGenerator described in Section 2.8.2.

2.8.6 DIET Webboard

Joint work with:

⋆ Nicolas Bard : CNRS. LIP Laboratory. Lyon. France.

DIET Webboard is a Java/Jsp web interface designed to manage all aspects of the Décrypthon Grid. Working with a MySQL database, it stores and updates in real time the state of each object in the Grid: storage spaces, data, jobs and results, computing nodes, requests to DIET, users...

DIET Webboard also implements the functionalities of GoDIET: it is capable of launching the whole DIET architecture, and also stops it when necessary. It includes a page for viewing the state of installed and running applications servers on the Grid. And finally, its code can be used to write a dedicated web interface for an application, such as the ones created for submitting jobs for Décrypthon applications.

The DIET Webboard's main page is a list of links to all the functionalities it implements. Managing the daemons, application examples, databases objects lists, monitoring and statistics, all these links are only enabled if the user has the right permissions.

The DIET Webboard application can run three daemons.

1. The DataBaseManager threads, which contains information for jobs submission and monitoring, it also checks periodically the database for errors, and finally it is able to submit again a job or a workunit with expired walltime for fault tolerance.

2. The GoDiet Thread, which is capable of launching and stopping the Diet platform.

3. The Diet test thread, this thread will launch many Jobs randomly on the Grid, for testing purpose.

The Diet Webboard has a strong connection with the Diet platform. Thus, it offers the possibility to check on the nodes running Diet SeD which applications are installed, which versions (according to their naming convention), and to detect which applications can run on each SeD. It also displays the list of disabled nodes and worker nodes used only through batch scheduler (LoadLeveler, OAR,...).

2.9 Conclusion

The aim of this chapter is to give an overview of the Diet architecture. We introduce the key points that made the success of Diet. The particular care we took in designing tools close to the user's needs, and the simplicity of usage of Diet, is the reason why Diet has been chosen by many Grid users. Moreover we gave an overview of external tools developed around Diet.

In the remainder of this document, we will focus on different parts of this architecture sorted by functionality. We started with the simple idea to have only a tool to play with your own scheduling algorithms and we ended with a real, and complete middleware, used in production for read scientific applications.

Chapter 3

DIET: Multi-Master Agent. Scalability from client side

Contents

As we have seen in Chapter 2, a naive approach is to build a DIET platform where every computer is managed by a unique Master Agent. From clients' point of view, this approach is not scalable, because each of them search to send a message to every computer that can resolve a specified problem, which generates an evaluation prediction on each computer. All the DIET clients try to access the same entry point of the platform. Moreover, this entry point is the root of the hierarchy. All requests from users and answers need to transit through this root. This is clearly the bottleneck. To avoid this bottleneck we decided to divide the hierarchy into many hierarchies and find a way to interconnect them.

Two versions were designed: the first one is static and use the Corba Multi-MA extension; and the second one is dynamic and uses the P2P approach.

3.1 Multi-hierarchies: The Corba Multi-MA extension

Joint work with:
- ⋆ Sylvain Dahan : University of Franche-Comté. LIFC Laboratory
- ⋆ Jean-Marc Nicod : University of Franche-Comté. LIFC Laboratory
- ⋆ Laurent Philippe : University of Franche-Comté. LIFC Laboratory

The multi-MA extension written by Sylvain Dahan under the supervising of Jean-Marc Nicod and Laurent Philippe to share computing resources between several sites. The Multi-MA allows the creation of several sets of computers. Each set is managed by a Master Agent. Those Master Agents are connected by a communication graph. When the user searches an available computer, he probes to a Master Agent, which searches a computer inside its own set of computers. If it could not find any available computer that can resolve the submitted problem, then it asks the other Master Agents if they have some computers available to resolve the problem. The choice of this algorithm comes from the assumption that we plan to optimize the deployment. That means clients should quickly connect to a correct hierarchy. We want to avoid to broadcast all requests, which would increasing significantly the number of messages. This way, requests are forwarded to a limited number of computers, unlike the monolithic approach where every computer resources is managed by a unique Master Agent. This allows several users to build a Grid platform, by sharing their resources, which is scalable with the number of simultaneous users.

Several DIET platforms are shared by interconnecting their Master Agent (MA). Clients ask for available SeD to their MA as usual. If the MA finds an available SeD which can resolve the problem, it returns a reference on it to the client. If it does not find a SeD, it forwards the request to other MAs which can also forward it to other ones and so on. When a MA finds a SeD which can resolve the client request, it returns its reference to the client's MA which returns the reference to the client. Then, the client can use this SeD to resolve its problem.

3.2 The P2P Multi-MA extension

Joint work with:
- ⋆ Cédric Tedeschi : ENS Lyon. LIP Laboratory. France.
- ⋆ Frédéric Desprez : INRIA. LIP Laboratory. France.

As Ian Foster mentioned in the book [83], P2P is a good way to deal with Grid environments. The heterogeneous and dynamic nature can be managed through a P2P approach. The aggregation of different independent DIET hierarchies (a multi-hierarchy architecture) could be managed using the P2P paradigm. We designed a P2P connection between Master Agents using the JXTA-J2SE toolbox [135] for on-demand discovery and connection of MAs. We called this prototype DIET$_j$.

Within DIET$_j$, the several MAs access each others' resources when processing a request, thus offering to the client an entry door to resources of several hierarchies put in common in a transparent manner. This results in avoiding the growing probability of bottleneck on one MA facing all requests thus increasing the scalability of the whole system.

The aim of DIET$_j$ is to dynamically connect together geographically distributed DIET hierarchies to gather services on-demand and improve the scalability of service discovery.

Dynamically connecting hierarchies for scalability. The clients are now given the ability to discover at time of requesting a service, one or several MAs, and thus connect the server with the best latency/locality.

Balancing the load among the MAs. The entry point for each client being dynamically chosen, the bottleneck on the previously unique Master Agent is now avoided. MAs are connected in a pure unstructured Peer-to-Peer fashion (without any mechanism of maintenance, routing, or group membership).

Gathering services at large scale. Whereas DIET hierarchies were unable to communicate together in the first version of DIET. With introduction of DIET$_j$ we provided the first Multi-MA behavior, services are now gathered when processing a request. Thus, providing clients a front door to resources of hierarchies put in common in a transparent way.

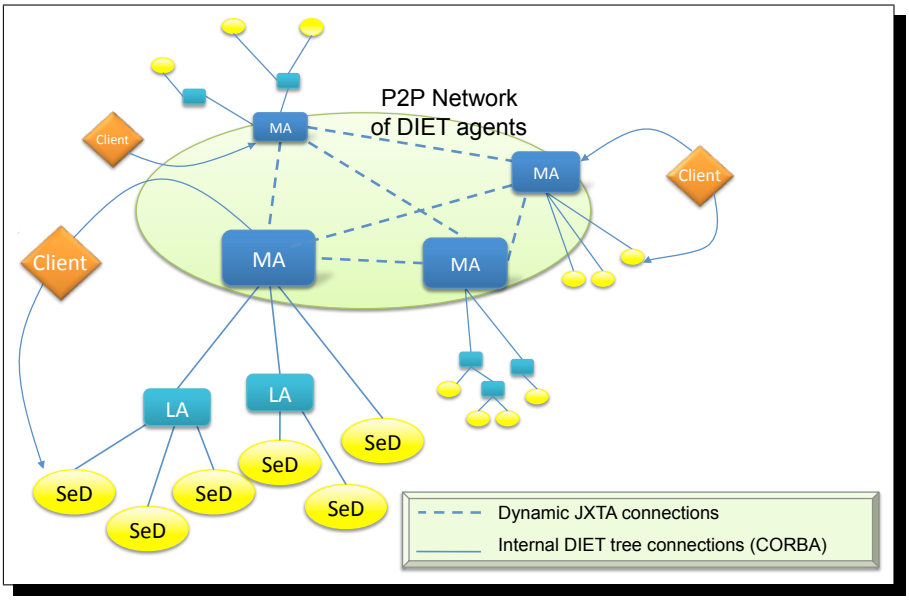


Figure 3.1: DIET$_j$ architecture.

3.2.1 DIET$_j$ Architecture

The DIET$_j$ architecture, shown in Figure 3.1, connects several DIET hierarchies through a JXTA network of Master Agents. The MA's internal architecture, shown on Figure 3.2 is divided into three parts.

- **The JXTA part.** The JXTA part of the MA is a peer on the JXTA virtual network. This part is its connection point to other Master Agents. This part is a java bytecode.
- **The DIET part.** The DIET part is the traditional DIET MA, root of a DIET hierarchy of Agents and Local Agents, allowing the discovery of servers that registered to this hierarchy. This part is based on libraries generated from the DIET C code.
- **The interface.** To cooperate, Java (JXTA native language) and C (DIET native API language) need an interface. We use the JNI technology allowing to call C functions from a Java program, and the data conversion between the two languages.

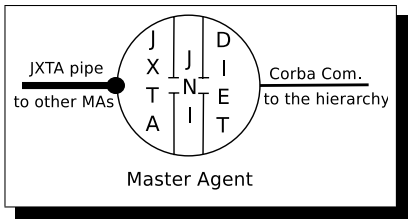


Figure 3.2: MA internal architecture.

3.2.2 The Multi-Master Agent system

One Multi-Master Agen (MMA) t is composed of all MAs running at a given time over the network and reachable from a first MA. The MA is able to dynamically connect to these other MAs. Each MA is known on the JXTA network by an advertisement with a name common to all of them ("DIET$_{MA}$") which is published at the beginning of its "life". This advertisement is published with a short lifetime to avoid Clients and other MAs to try to bind to an already stopped MA, and thus easily take into account the dynamicity of the platform.

During the Master Agent loading time, the JXTA part loads the DIET part via JNI, periodically re-publishes its advertisement, while waiting for requests. When receiving a client's request, the DIET part submits the request to its own hierarchy. If the submission to the DIET hierarchy retrieves no SeD's references with the required service, the JXTA part builds a multi-hierarchy by discovering other MAs (thanks to their JXTA advertisements) and propagates the request to them. When the JXTA part has received responses from all other MAs or when a timeout is reached, the response is sent back to the client which is not aware that a multi-hierarchy has been temporarily built.

3.2.3 Dynamic Connections

Dynamic connections between the MAs allow to transparently perform the service discovery in a dynamic large scale multi-hierarchy, using JXTA advertisements. The communication between the agents inside one hierarchy are still static as we suppose that small hierarchies are installed within each administrative domain. At the local level, performances are not very variable and new elements are not frequently added.

3.3 Traversing the Multi-Hierarchy

We now discuss approaches and algorithms implemented for propagating the clients' requests and gather information about servers of several hierarchies.

3.3.1 Approach

Discovering the MAs, then discovering the servers

It is important to note that the multi-hierarchy construction is divided into two parts.

1. **peers discovery**. The first step aims at discovering MAs reachable on the network, thanks to the JXTA discovery process. But once a peer has been discovered, i.e., got its advertisement (mainly containing its name and its address, under the shape of an input pipe advertisement), you still need to establish a connection with it.

2. **service discovery**. The second step consists in exploring the multi-hierarchy composed of the MAs discovered in the first step, looking for the requested service inside the DIET hierarchies.

JXTA discovery mechanisms

JXTA 2.x provides a hybrid mechanism based on DHT [165] and random walk to achieve the discovery of advertisements (e.g., advertisement named "DIET$_{MA}$"). Again, we choose not to use the hybrid DHT mechanism to avoid its maintenance overhead, its lack of exhaustiveness (when failing retrieving the advertisement by the hash function, it uses a less-efficient "walking" method) and cope with the unstructured fashion of the multi-hierarchy designed.

Thus, we first use the JXTA discovery mechanism based on flooding among the peers. Once the MA's references are obtained, an algorithm optimizing the traversal of the multi-hierarchy (the MAs graph) is used to connect MAs together and propagate the request through the multi-hierarchy.

3.3.2 Implementations

The propagation of the request in the DIET$_j$ multi-hierarchy (i.e., between the MAs) has been implemented with two algorithms.

Propagation as an Asynchronous Star Graph Traversal

The propagation has first been implemented as an intuitive asynchronous star graph traversal. One MA r which found no SeD providing the service requested by a client in its own hierarchy, discovers other MAs with the JXTA discovery process. Then, it forwards the request in an asynchronous way to all the MA previously discovered, using a simple JXTA multicast pipe instruction. On receipt of the forwarded request, each MA collects the servers able to solve the problem in its own hierarchy, and sends back the response to r that collects and merges responses to create the final response message, that it sends back to the client. Using this first algorithm, the propagation systematically builds a star graph, the MA initiating the propagation being the root of the star graph. We called this algorithm "STAR$_{async}$".

Propagation as an Expanded Version of the Asynchronous PIF Scheme

The propagation has also been implemented using an asynchronous version of the **P**ropagation of **I**nformation with **F**eedback scheme (PIF), to have an unstructured, efficient and adaptative multi-hierarchy traversal. A complete description of the basic PIF can be found in [60, 151]. Figure 3.3 describes a scenario of propagation in a DIET multi-hierarchy, applying the two following phases:

1. The **Broadcast phase**: The MA that received the request from the client (and is unable to find a server providing the requested service) initiates the wave, and so is the root r. As in the STAR$_{async}$ algorithm, it forwards the request to all other MAs it has previously discovered. Let M_r be the set of discovered MAs. r then waits for responses of MAs in M_r. The MA

that sent the request to it becomes its parent. Of course, m collects the servers to solve the problem described in the request in its hierarchy. Finally, m propagates the request in its turn to the MAs in M_r (that it knows from its parent), except those that are the way taken by the request to reach m from r. Thus a time optimal tree rooted at r is built.

2. The **Feedback phase**: r waits for the responses of M_r during a finite time using a timeout. The MAs in M_r send the enabled servers found in their hierarchy back to their parent,t and, when receiving a response from a child, send the response to their own parent.

PIF scenario Let us have a look at Figure 3.3. The MA that received the request from the client found no SeD providing the requested service in its own hierarchy. After having discovered other MA, it initiates the wave (1). Some MAs have received the propagated request. They forward it in their turn, and initiate the asynchronous feedback phase (2). All MAs have received the request. A spanning tree is built. The feedback phase goes on and ends. The connections opened during this phase depends on the traffic load encountered during the broadcast phase, allowing an optimal feedback phase (3).

Quick analysis of the PIF scheme

Let us call this algorithm "PIF$_{async}$". Note that PIF$_{async}$ builds an *on-demand optimal tree* for a given root for each request, thus balancing the load among the MAs graph as the number of requests increases and also avoiding overloaded links. It was shown in [145, 151] that in asynchronous environments, the PIF scheme is the fastest possible to reach every network nodes, messages following the fastest links during the broadcast phase. In other words, the dynamic tree built during the propagation is time optimal. It provides fault tolerance, because of the several retransmissions achieved by this algorithm and thus the several attempts to reach each MA. The number of messages can be very important ($O(n^2)$ in the worst case). Note that algorithm STAR$_{async}$ also provides a PIF scheme. However, it is not an adapting scheme, messages always follow the same links, ignoring their heterogeneity and communication load.

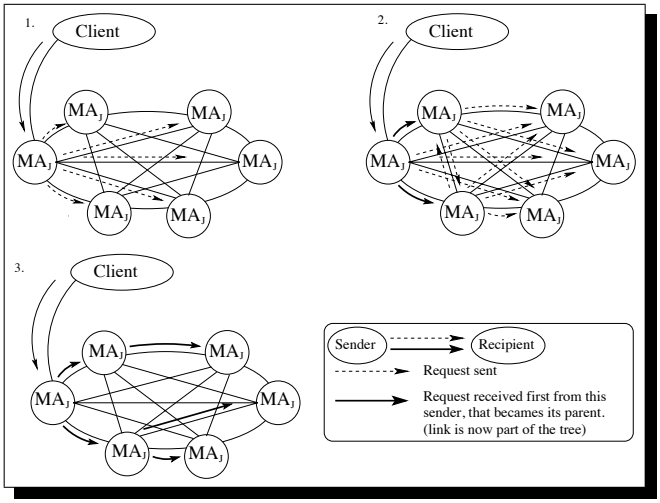


Figure 3.3: Propagation scenario in a DIET multi-hierarchy.

3.4 A new mechanism for Service Discovery in P2P network

Joint work with:
⋆ Cédric Tedeschi : ENS Lyon. LIP Laboratory. Lyon. France.
⋆ Frédéric Desprez : INRIA. LIP Laboratory. Lyon. France.
⋆ Ajoy K. Datta : University of Nevada. Las Vegas. USA.
⋆ Franck Petit : Université de Picardie Jules Verne. MIS Laboratory. Amiens.

Beyond the MMA P2P architecture we decided to provide a service Discovery in P2P network. From a need of DIET, we have extended this work to a general purpose. The core element of this architecture, indexing the services, is a distributed trie. It was called DLPT, for Distributed Lexicographic Placement Table or DPT for Dynamic Prefix Tree as referred to as in several papers [59] for instance. The DLPT is a stand alone system. It even became the base of a platform in the SPADES ANR (see Section 3.4.3) that I lead. We give here an overview of this system.

The resource discovery in P2P environments has been intensively studied. Although DHTs [144, 147, 158] were designed for very large systems, they only provide rigid search mechanisms. A great deal of research went into finding ways to improve the retrieval process over structured peer-to-peer networks. Peer-to-peer systems use different technologies to support multi-attribute range queries [29, 121, 150, 153]. Trie-structured approaches outperform others in the sense that logarithmic latency (or constant if we assume an upper bound on the depth of the trie) is achieved by parallelizing the resolution of the query in several branches of the trie.

Among trie-based approaches, Prefix Hash Tree (PHT) [142] dynamically builds a trie of the given key-space (full set of possible identifiers of resources) as an upper layer mapped over any DHT-like network. Fault-tolerance within PHT is delegated to the DHT layer. Skip Graphs, introduced in [21], are similar to tries, and rely on skip lists, using their own probabilistic fault-tolerance guarantees. P-Grid is a similar binary trie whose nodes of different sub-parts of the trie are linked by shortcuts like in Kademlia [125]. The fault-tolerance approach used in P-Grid [68] is based on probabilistic replication.

3.4.1 DLP-Tables

The DLPT (Distributed Lexicographic Placement Table) is the architecture that we have recently developed and studied in [54]. This approach, initially designed for the purpose of service discovery over dynamic computational Grids and aimed at solving some drawbacks of similar previous approaches, is a two layer architecture. The upper layer is a prefix tree maintaining the information about available services. Each node of this tree maintains the information about services sharing a particular name or (**key**), used to label the node. This tree is built dynamically as services are registered by some servers of the computational platform, as illustrated by Figure 3.4. Nodes storing some services' references (and labeled by actual names of services) are grey-filled, the others, created to ensure the consistency of the structure and the routing of queries, are labeled by **virtual** keys. (a) First a `DGEMM` is declared. (b) A `DTRSM` is declared resulting in the creation of their parent, whose label is their greatest common prefix, i.e., `D`. (c) Finally, a `DTRMM` is declared and the node `DTR` is created. This tree is mapped onto the lower layer, i.e., the physical network, for example, using a distributed hash table. An advantage of this technology is its ability to take into account the heterogeneity of the underlying physical network to build a more efficient tree overlay, as detailed in [55]. Simulations available in [54] show that, using different data sets, approximately 1/3 of nodes are labeled by **virtual** keys. In other words, 1/3 of the nodes has the sole purpose to

ensure the consistency of the architecture, 2/3 of nodes storing actual services' references.

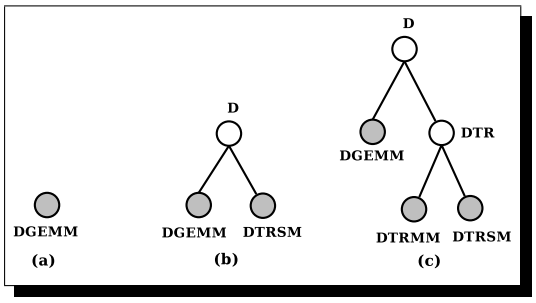


Figure 3.4: Construction of the prefix tree in DLPT approach.

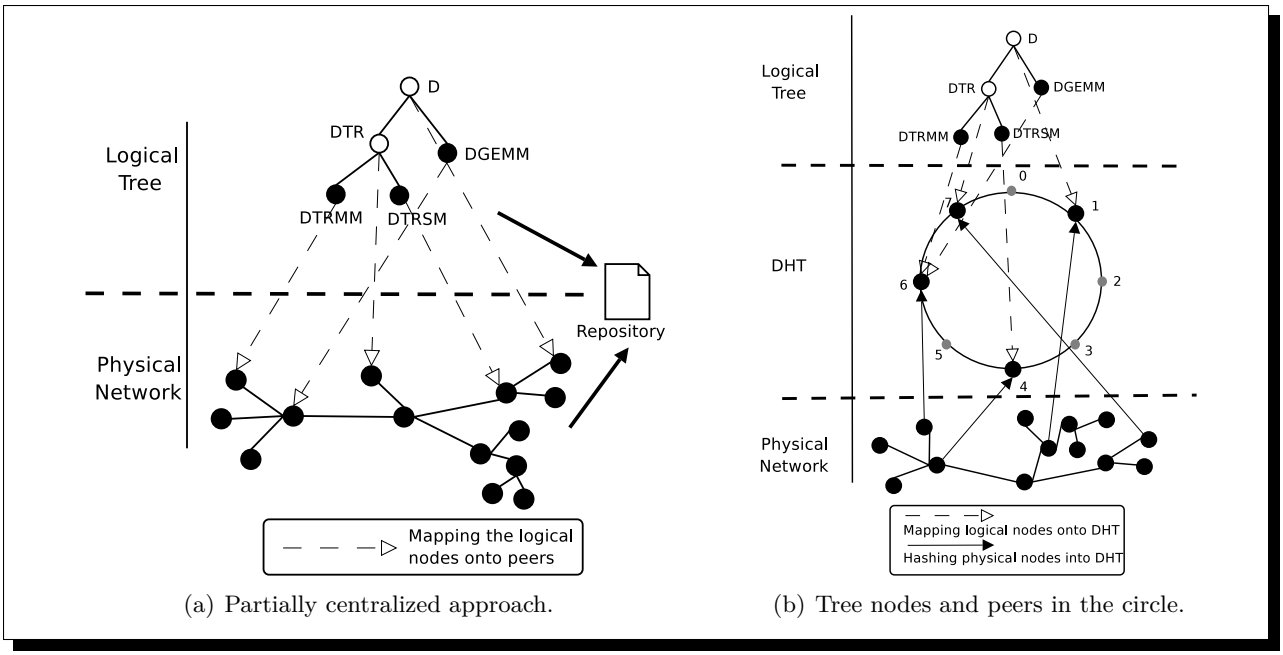
When a discovery request sent by a client enters the tree, on a random node, the request moves upward until reaching a node whose subtree contains the requested node and then moves downward to this node. The DLPT system supports range queries and automatic completion of partial search strings. In its original design [54], the tree is mapped over the network using a distributed hash table.

Mapping design for the DLPT

How the logical entities (nodes of the tree, referred to as nodes in the following) were mapped on the processors of the network (referred to as peers henceforth), i.e., the question: which peers executes which node? was left aside. We now briefly discuss how to map the trie onto peers. We describe two architectures we have defined to implement the DLPT in a real environment.

Centralized approach. In a first simple approach illustrated in Figure 3.5(a), the mapping relies on a central device. Obviously, this device is only used for the mapping and does not have any role when processing insertion or discovery requests. The peers, when connecting the network, join the DLPT by registering themselves to this central device. When a new logical node is created, a peer is randomly chosen to host it among the peers that registered to this central repository. This approach is intuitive, easy to implement, but as in every centralized systems, the central device is a single point of failure, even if duplicating it. This led us to define a distributed architecture.

Distributed approach In a second approach, a DHT structures the network and a peer can be chosen by hashing the node ID, as illustrated on Figure 3.5(b). Indeed, any DHT could be used. An issue we do not consider in this manuscript is related to load balancing. Obviously, using a DHT to uniformly distribute the logical nodes on the peers do not achieve an efficient balancing of the workload. The load of a node depends on the popularity of services and its depth in the trie (nodes close to the root are more solicited than leaves when routing requests). DHTs make two common assumptions. First, they consider that the capacities of peers are homogeneous, which can not be ensured on real Grids. They also assume that each data item has the same probability to be requested. We rely on several recent works addressing the heterogeneity of both the capacity of peers and popularity of nodes inside DHTs [88, 104, 120].



(a) Partially centralized approach.

(b) Tree nodes and peers in the circle.

Figure 3.5: Architecture of DLPT mapping

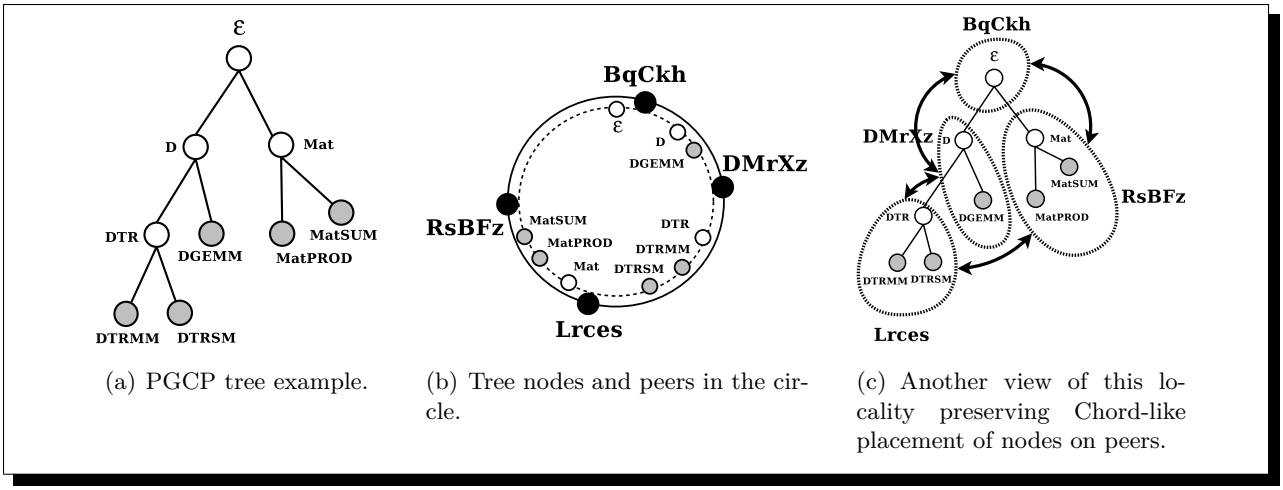
The decentralized approach is better for scalability, thus we choose the second one, based on the DHT, however this leads to two drawbacks:

- The presence of an underlying system like a DHT, leads to the need to maintain two layers (one DHT, and one tree over the DHT), making the maintenance cost of this two-layer architecture extremely high.
- The routing scheme presented in the previous section, as well as the heterogeneity on both popularity of keys and capacity of peers could lead to an unbalanced distribution of the load and thus create bottlenecks on different peers.

Tackling these two drawbacks, we now focus on a mapping scheme having several properties:

- This scheme is self-contained (avoid the need for a DHT or any extra tool) and maintain the tree indexing the services while mapping it on the peers.
- On top of this protocol, it uses some load balancing heuristics based on local maximization of the throughput i.e., the number of requests processed by the service discovery system. This heuristic is based on existing approaches for load balancing within DHTs.

From now on, we consider that IP addresses of peers are still hashed, but the hash function to find the peer hosting one node is locality-preserving. In other words, the nodes' labels are not hashed but nodes are just placed on the peer whose id is the smallest higher than the label of the node, assuming that we hash IP addresses of peers by using a hash function which has values within the set of possible labels of nodes. As a simple example, consider the PGCP tree given in Figure 3.6(a). The mapping achieved is similar to Chord [158] (consistent hashing) in the sense



(a) PGCP tree example.

(b) Tree nodes and peers in the circle.

(c) Another view of this locality preserving Chord-like placement of nodes on peers.

Figure 3.6: Example of mapping

that each node is run by the successor peer of the node's label. Have a look at Figure 3.6(b) The tree nodes (transparent and grey-filled on the internal dashed circle) and the peers (big and black filled on the external circle with an identifier randomly picked in the set of possible strings) are in the same circle identifier space. Now, each peer is supposed to run nodes whose labels fall in the range between itself and its predecessor peer. For instance, peer DMrXz runs nodes D and DGEMM. Nodes are again connected through the tree links (that do not appear on Figure 3.6(b) for clarity), peers are also connected, on a basic ring. Another view of this locality preserving Chord-like placement of nodes on peers is given by Figure 3.6(c). Peers (pseudo-ellipses in dashed lines) are connected in a ring (bold arrowed lines), each peer running the set of nodes inside its ellipse.

Load balancing for the DLPT

The DLPT focus on P2P architecture beyond the scope of the forest of DIET trees. Thus we have designed this structure to be compliant with high dynamic nature and to support a high level of scalability. In this context, a load balancing is required to ensure a good performance even when the number of nodes grows. We introduce here the main idea of the solution we proposed for the DLPT.

Each peer runs a set of nodes. As detailed before, the routing follows a top-down traversal. Therefore, the upper node is, the more times it will be visited by some request. Moreover, due to the sudden popularity of some service, the nodes storing the corresponding keys, independently from their depth in the tree, may become overloaded. The technique we present now deals with this issue by maximizing the aggregated throughput of two consecutive peers, i.e., the number of requests these two heterogeneous peers will be able to process. This is achieved by periodically redistributing the nodes on the peers, based on recent history.

Let us consider one particular peer S to illustrate the load balancing process. This is periodically triggered on S. Let P be the predecessor of S. Refer to Figure 3.4(a). CS and CP refer to their respective capacities, i.e., the number of requests they are respectively able to process during one given period of time. Note that the peers capacity does not change over time. At the end of one

period, some peers send the number of requests they received during this time unit, for each node it runs, to its successor. Here, S has information on the loads of the nodes run by P, and obviously knows its own load history. Assume that, during last time unit, the set of nodes run by S and P were respectively NS and NP and that each n of the union of NS and NP has received a different number of requests, denoted by ln. Then, the load of S, denoted by LS during the ending period was the sum of the loads of the nodes it runs. Finally, we easily see that, during this last period, the number of satisfied requests (or throughput T), i.e., requests that were effectively processed until reaching its destination is T = min(LS, CS) + min(LP , CP).

Starting from this knowledge, i.e., the load of every nodes n in NS and NP, we want to maximize the throughput of the next period of time. To do so, we need to find the new distribution (NS, NP) that maximizes T, assuming the load distribution in two consecutive periods will not be the same, but close. The number of possible distributions of nodes on peers is bounded by the fact that nodes identifiers can not be changed, in order to ensure the routing consistency and a way to retrieve services. The only parameter that we can change is the identifiers of peers. Then, as illustrated on Figure 3.7, finding the best distribution is equivalent to finding the best position of P moving along the ring, as illustrated by arrows on Figure 3.7(b). The time and extra space complexity of the redistribution algorithm is clearly in O(card(NS) + card(NP)). In other words, even if periodically performed, the MLT heuristic has, locally, a constant communication cost and a time complexity linear in the number of nodes between two local peers. An example of the result of this process is given in Figure 3.7(c), where, according to the previous metric, the best distribution is 3 (weighted) nodes on S, and 5 (weighted) nodes on P. This heuristic is henceforth referred to as MLT (Max Local Throughput).

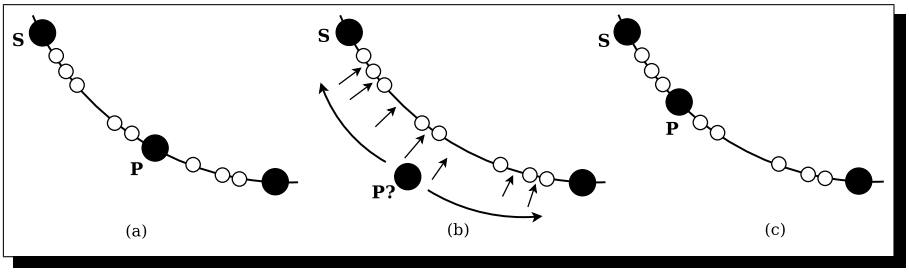


Figure 3.7: One local load balancing step.

Fault tolerance mechanism in the DLPT

The basic design of the DLPT presented in [54] has the ability to greedily take into account the heterogeneity of the underlying physical network to make a more efficient tree overlay. However, in [54], the fault tolerance is still addressed by replication of nodes and links of the tree. To summarize, the fault tolerance issue is mostly either ignored, delegated or based on replication. In [48], we provided a first alternative to the replication approach. The idea was to let the trie crash and to reconnect and reorder the nodes rounds build is the x However, this protocol assumed the validity of subtries being reordered, thus limiting the field of initial configurations handled and repaired. In [45] we present a new protocol able to repair any labeled rooted tree to make a valid greatest common prefix tree and thus offer a general mechanism to maintain distributed tries.

In the self-stabilizing area, some investigations take interest in maintaining distributed data structures. The solutions in [98, 99] focus on binary heap and 2-3 trees. Several approaches have also been considered for a distributed spanning tree maintenance [77]. In [97], a self-stabilizing spanning tree construction is also proposed for a model, introduced in the same paper. The proposed model introduces mechanisms that fit large scale systems. Roughly speaking, their model is based on the classical message-passing model in which a mechanism for resource discovery called oracle is added. In [25], the authors presented the first snap-stabilizing distributed solution for the Binary Search Tree (BST) problem. Their solution requires $O(n)$ rounds to build the BST, which is proved to be asymptotically optimal for this problem in the same paper.

In [52] we have presented the first snap-stabilizing greatest common prefix tree for peer-to-peer systems. It provides an alternative to tree-structured peer-to-peer networks suffering from the high cost of replication and an innovating way to implement fault tolerance over large distributed systems. We have developed an optimal algorithm in terms of stabilization time (snap-stabilizing). It requires an average number of rounds proportional to the height of the tree, thus providing a good scalability. This result has been confirmed by simulation using actual data sets in [52].

3.4.2 Multi-Attribute Searches

As illustrated on Figure 3.8, supporting multi-attribute queries is achieved using a simple extension of the previous algorithms to a multi-dimensional system in which each dimension (or `type` of attribute is maintained by a distinct overlay. Then, the value/address of a service having several attributes is stored in each overlay. For multi-attribute requests, like `{DTRSM, Linux*, PowerPC*}`, the client sends three independent requests. The request on `DTRSM` will be sent to the `services' names` tree, `Linux*` to the system tree and `PowerPC*` to the processor tree. Requests are independently processed within each tree and the client asynchronously receives the values and finally intersects the sets of locations obtained to only keep answers matching the three values requested.

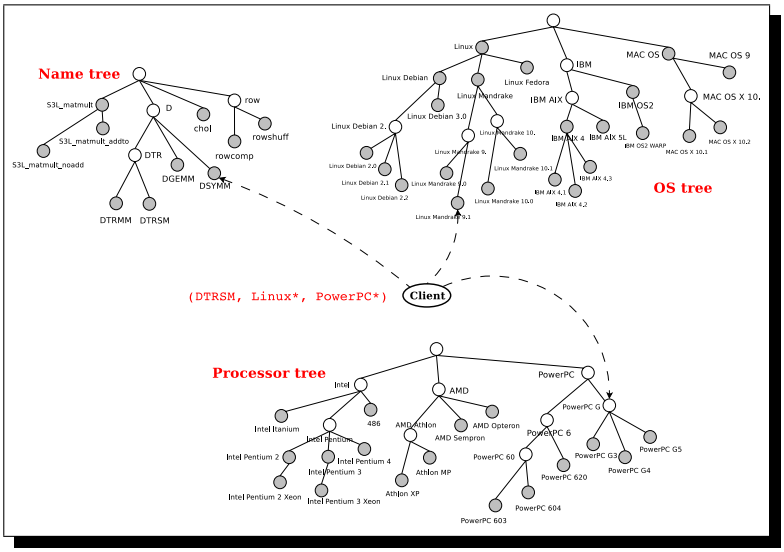


Figure 3.8: Processing a multi-attribute query.

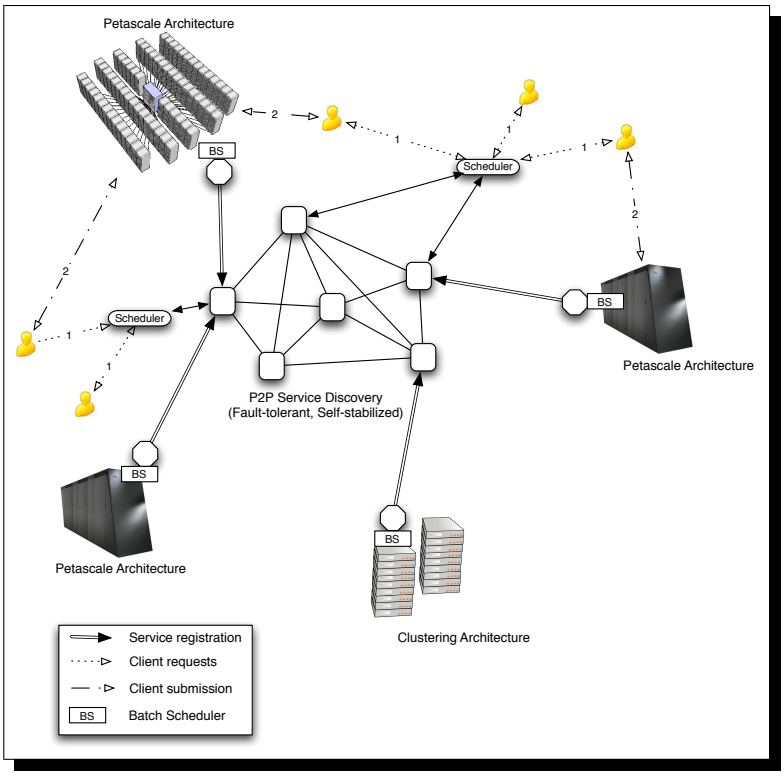


Figure 3.9: The architecture of the platform SPADES based on DLPT.

3.4.3 SPADES: Emerging Platform based on DLPT

Joint work with:

⋆ Cédric Tedeschi	: ENS Lyon. LIP Laboratory. Lyon. France.
⋆ Florent Chuffart	: INRIA. LIP Laboratory. Lyon. France
⋆ Haiwu He	: INRIA. LIP Laboratory. Lyon. France
⋆ Franck Petit	: Université de Picardie Jules Verne. MIS Laboratory. Amiens. France.
⋆ Frédéric Suter	: CNRS. IN2P3. Lyon. France.

Today's emergence of Petascale architectures and evolutions of both research and production Grids increase a lot the number of potential resources. However, existing infrastructures and access rules do not allow to fully take advantage of these resources.

One key idea of the SPADES [1] project (ANR project) is to propose a non-intrusive but highly dynamic environment able to take advantage of available resources without disturbing their native use. In other words, SPADES' vision is to adapt the desktop Grid paradigm by replacing users' computers at the edge of the Internet by volatile, but highly efficient resources. Batch schedulers (BS) manage these volatile resources, with reservation mechanisms. Access to these machines may be limited in time and susceptible to preemption (best-effort mode). One of the priorities of SPADES is to support platforms at a very large scale. Petascale environments are in consequence particularly considered. However, these next-generation architectures still suffer from a lack of expertise for an accurate and relevant usage. One of SPADES' goals is to show how to take

[1] `http://graal.ens-lyon.fr/SPADES`

advantage of the power of such architectures. Another challenge is to provide a software solution for a service discovery system able to face a highly dynamic platform. To reach this goal the system used is the DLPT described above. This system will be deployed over volatile nodes and thus must tolerate “failures". The implementation of such an experimental development also leads to the need for an interface with batch submission systems able to make reservations in a transparent manner for users, but also to be able to communicate with these batch systems in order to get the information required by our schedulers. SPADES will propose solutions for the management of distributed schedulers in Desktop Computing environments, coping with a co-scheduling framework (see Figure 3.9).

3.5 Conclusion and Future Work

In this chapter, we have presented DIET$_j$, the first extension of a Network-Enabled Server system taking into account the dynamic and heterogeneous nature of today's platforms on which Grids will inexorably take place. The use of JXTA and the asynchronous PIF algorithm shows an efficient on-demand discovery of available servers at a large scale. Beyond this DIET functionality we have developed a new service discovery called DLPT based on tree. This system has been improved with load balancing and self-stabilizing mechanisms to ensure fault-tolerance. Armed with this concept we have designed an emerging platform around co-scheduling, service discovery and large amounts of resources provided through different infrastructure such as Petascale architecture.

Chapter 4

DIET: Resources Management. Scalability and Heterogeneity from server side.

Contents

DIET was designed for high performance computing. In this area, resources are shared between users. But each user would rather to have resources in a dedicated environment for different reasons

- avoid conflict between applications,
- take benefit of the maximum power, memory and storage until resources are freed,
- increase reproducibility,
- from the point of view of the resource providers, the dedicated paradigm allows to control the resource access, thus provider can apply their own policy.

Parallel Grid resources (parallel machines or clusters of workstations) are generally managed by a reservation batch system (a.k.a., LRMS for Local Resource Management System) such as OAR [40], SGE [86], LSF [102], Loadleveler [139]. Such a system is responsible for managing the submitted jobs, locating and allocating the required resources. It accepts user submission scripts which must normally contain a variety of information including the requested number of resources and the amount of time needed for the reservation (walltime).

4.1 Parallel systems

Basic parallel systems are surely the less deployed in actual computing Grids. They are usually composed of a gateway where clients log in, and from which they can log on numerous nodes and execute their parallel jobs, *without any kind of reservation (time and space)*. Some problems occur with such a use of parallel resources: (1) multiple parallel tasks can share a single processor, hence delaying the execution of all applications using it; (2) during the deployment, the application must at least check the connectivity of the resources; (3) if performance is wanted, some monitoring has to be performed by the application.

4.2 DIET LRMS Management

Joint work with:

⋆ Yves Caniou	: University of Lyon Claude Bernard. LIP Laboratory. Lyon. France.
⋆ Frédéric Desprez	: INRIA. LIP Laboratory. Lyon. France.
⋆ Jean-Sébastien Gay	: ENS Lyon. LIP Laboratory. Lyon. France.

A rule of DIET is to hide the Grid complexity to the user. That means we should provide *transparent access* to parallel resources for the user. It must choose the best parallel resource that suits the request, eventually provide for the parallel malleable tasks [1] the right number of processors, provide the corresponding walltime, and submit this information to the LRMS in an automatically built script in the language of the reservation system. Indeed, as the user does not need to know where his/her job is executed (he does not need to know the computation availability of a given host, the memory availability, etc.), such a script has to be produced by the middleware in place of the user. The DIET environment must implement such mechanisms to provide transparent access to parallel resources and minimize at the same time the completion time of the parallel task. These developments have been made by the efforts provided under the guidance of Yves Caniou.

4.2.1 Sequential and parallel requests

Two extensions (at the client and server level) of the DIET API provide means to request exclusively sequential services, parallel services, or let DIET choose the best implementation of a problem for efficiency purposes (according to the scheduling metric and the performance function). DIET provides an API for the client, thus the client can explicitly call a parallel call `diet_parallel_call(diet_profile_t *profile)`. Moreover if the user wants to choose the amount of required resources, he can be using the `diet_profile_set_nbprocs()` function.

4.2.2 Transparently Submitting to LRMS

DIET can submit jobs to LRMS including Loadlever, OAR (v1.6 and v2.x) and PBS. We plan to complete integration with many other such as the WMS system used in the EGEE [119] and SLURM [103].

As we have said, the DIET parallel/batch API provides several functions on both client and server side. On the client side, the client can explicitly ask for a sequential/parallel computation of its job, but otherwise and whenever possible, DIET will choose the best available allocation among sequential/parallel resources (more information will be given to this process in the Chapter 8). On

[1] A malleable task is a computational unit which may be executed on any arbitrary number of processors.[130]

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

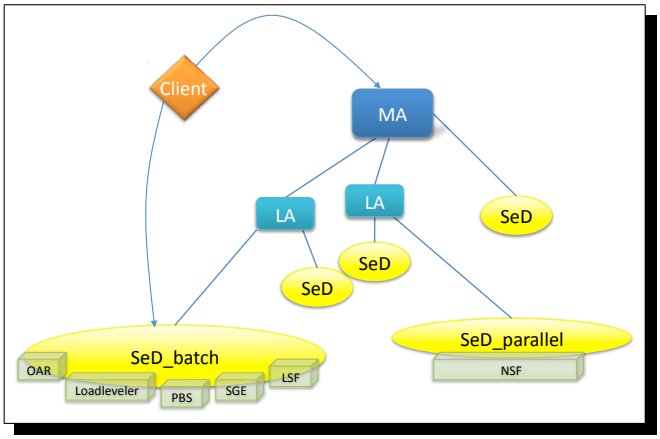


Figure 4.1: The DIET architecture with parallel and LRMS support.

To conclude this first part of this chapter, we have now three kind of SeD available with DIET for different usage. Figure 4.1 is an example of DIET deployment using different kind of SeD: The basic SeD, the parallel SeD and the SeD-batch are involved. The next generation of the LRMS are based on virtualization. The concept is not new as it exists since the 60's as mentioned in [91]. Nowadays, resource reservation can be done through the deployment of Virtual Machines (VM) such as VMware [172] or Xen [23]. The VM are a technical solution to open the Cloud platform to the users. The VM may provide an homogeneous-like environment and offer a certain level of security based on the sandbox paradigm. In the next section we will see how DIET can work with Cloud platforms.

4.3 Cloud resources management

Joint work with:
⋆ Frédéric Desprez : INRIA. LIP Laboratory. Lyon. France.
⋆ Adrian Muresan : ENS Lyon. LIP Laboratory. Lyon. France.

Over the last years, Internet Computing and storage have considerably evolved from small isolated nodes to large-scale Cluster-like architectures driven by efficiency and scalability needs which we now know under the name of "Clouds" [19, 168]. They aim at being dynamically scalable and offer virtualized resources as service over the Internet. Usually solutions deployed in Clouds are web browser targeted and are load balanced when it comes to computational power and storage. Clouds can also be used in more computational-intensive domains by using them as scalable computational resources.

Grid platforms offer a large benefit when it comes to computational power, yet they have a drawback caused by their scalability. To get the best of both worlds, a Grid middleware can be used to manage and harness raw Cloud computational resources.

The current chapter will only scratch the surface of this interesting topic. We offer a proof-of-concept of using a Cloud system as computational resource through a Grid middleware. We demonstrate the use of EUCALYPTUS [134], the open-source Cloud, as a resource for DIET.

4.3.1 EUCALYPTUS

In the world of open-source Cloud Computing, EUCALYPTUS [133] is a pioneer. It relies on commonly-available Linux tools and simple Web Service technologies and is compatible with the Amazon Elastic Computing Cloud (EC2) SOAP interface [2].

The EUCALYPTUS platform has a three-level hierarchical architecture. At the top of the hierarchy lies the Cloud Controller node (CLC). Its role is to coordinate the Cloud as a whole and to handle client Cloud management requests. This is the only node that is responsible for decision-making.

Halfway between the top and bottom of the hierarchy lies the Cluster Controller (CC). It is responsible for keeping track of resource usage in its Cluster.

At the bottom of the hierarchy lies the Node Controller (NC). Each physical machine that is to be a computing machine needs to have the NC service running. The NC has two main responsibilities: monitoring resource usage and managing virtual resources.

To achieve virtual resource management, EUCALYPTUS uses Xen [23] and/or KVM [111] virtualization technologies. These offer great benefits when dealing with scalability and application isolation but also have drawbacks due to the necessary start-up and shutdown time of the virtual machines.

We have chosen to use the SOAP interface that EUCALYPTUS provides. This is more flexible than the query interface (the query string has a limited size) and offers better security because it implements the WS-Security standard with an asymmetric RAS key pair. As a result we have successfully managed to programmatically manipulate EUCALYPTUS virtual resources and gain direct access to them once instantiated. On the basis of the above results we are ready to integrate EUCALYPTUS as a resource into DIET.

4.3.2 DIET over a Cloud

With the question of how to harness EUCALYPTUS as a DIET resource in mind we need to consider the architectures of both systems in order to find a suitable answer. From a high-level perspective we have several plausible solutions that differ by how much of the architectures of both systems overlap or are included one in the other. To be more precise, we can consider the following two scenarios and any other scenario that is logically between the two.

DIET is completely outside of EUCALYPTUS: In this scenario the DIET and EUCALYPTUS architectures do not overlap at all in the sense that all DIET agents or SeDs run separately with respect to the EUCALYPTUS controllers. The DIET SeD requests resources (compute nodes) to EUCALYPTUS when needed and uses the resources directly (bypassing EUCALYPTUS broker) once they are ready. In this scenario scalability is limited because of the fixed DIET architecture that cannot scale easily, but the number of compute nodes takes full advantage of EUCALYPTUS's scalability.

DIET is completely included in EUCALYPTUS: The DIET architecture is virtualized inside EUCALYPTUS. This scenario is the other extreme of the previously stated one since DIET agents and SeDs are virtualized and instantiated on-demand. It is obvious that this scenario offers more flexibility because one can configure the amount of physical resources that are allocated to EUCALYPTUS virtual resources. This is also a more scalable approach because of the on-demand way of use that is typical to Cloud platforms.

The simplest and most natural scenario from the perspective of DIET is to treat EUCALYPTUS as a new type of resource allocator and scheduler inside the SeD since DIET is easily extensible in this direction.

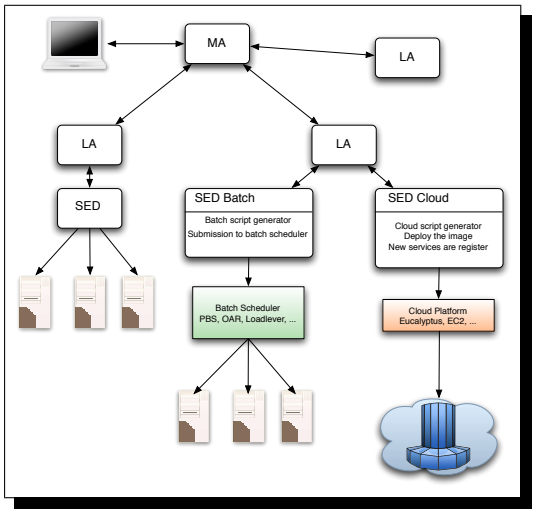


Figure 4.2: DIET architecture with three different kinds of server daemons, including Cloud server.

Figure 4.2 shows the architecture of DIET with three kinds of server daemons:

SeD: The basic SeD encapsulates a computational server. The information stored on a SeD is a list of data available on it, the list of problems that it can solve and performance-related information.

SeD Batch: is an upgraded version of the basic SeD. This SeD has the capability of submitting requests to different Batch Schedulers (BS) without the user having to know how to submit to the underlying BS.

SeD Cloud: based on the same idea that the previous one: this version provides the capability to use Cloud resources through the API of the Cloud platform. Knowing which image is associated to which service, the SeD can then automatically deploy it on the selected number of compute nodes.

Handling of a service call is done in three steps.

Obtain the requested virtual machines. This first step involves requesting the virtual machines to EUCALYPTUS by using its SOAP API. The SeD Cloud contains a mapping of services

to virtual machines that the user is not aware of (and does not need to be aware of). The start-up of virtual machines is a time consuming operation and is best done asynchronously with the request. As a result, the SeD Cloud polls EUCALYPTUS to receive a positive or negative reply related to the number of virtual machines requested. If the reply is positive then the SeD Cloud can proceed to the second step in handling the request.

Execute the MPI service on the instantiated virtual machines. The end result of the previous step is a list of addresses of the instantiated virtual machines. Having this low-level information, the SeD Cloud can now initiate a parallel request such as a MPI request on the machines. The result of the calculation is returned and a reply is formed for the service call.

Terminating the virtual machines. The SeD Cloud initiates another SOAP request to EUCALYPTUS for the termination of the instantiated virtual machines.

4.4 Conclusion and Future work

In this chapter we have shown how DIET can manage resources. We have seen different methods to address them: basic, parallel, through a LRMS or using a Cloud environment. The modular conception of DIET helps the developer to add new LRMS or new Cloud environments easily. A future work in this topic will be to add the virtual SMP architecture using for example the Kerrighed technology [129] as described in Figure 4.3. Kerrighed virtualizes a cluster to an SMP and DIET could control and updates the Kerrighed resources related to the platform load. A communication layer between Kerrighed and DIET schedulers through the **SeD Kerrighed** is required to provide an efficient and dynamic management of the resources. DIET will add or remove nodes to the Kerrighed platform to grow or reduce the SMP computer on demand.

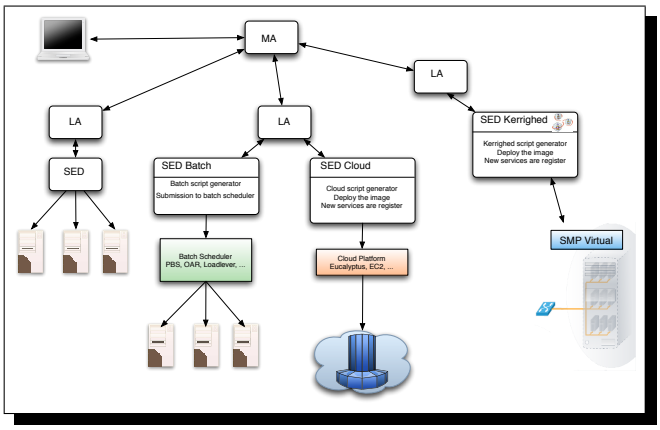


Figure 4.3: DIET architecture with Virtual SMP as resources.

Chapter 5

Hierarchical Deployment Planning for DIET

Contents

While middleware designers often note that the problem of deployment planning is important, only a few algorithms exist [44, 106, 108] for efficient and automatic deployment planning. Questions such as "which resources should be used?", "how many resources should be used?", "what arrangement should be used", and "should the fastest and best-connected resource be used for a scheduler or as a computational resource?" remain difficult to answer. In [169] the authors state the need for adaptive middleware technologies for Grid environments. Technologies that can select resources from the Grid to a better fit to the users' expectations, and that can do proper configuration of the selected resources, are needed.

Related to the DIET architecture we focused on hierarchical arrangements. A hierarchy is a simple and effective distribution approach, and has been chosen by a variety of middleware environments as their primary distribution approach [47, 67, 95, 149]. Before trying to optimize deployment planning on arbitrary, distributed resource sets, we target a smaller sub-problem that has previously remained unsolved: what is the optimal hierarchical deployment on a cluster with hundreds to thousands of nodes? This problem is not as simple as it may sound: one must decide how many resources should be used on the whole, how many should be dedicated to scheduling or computation, and which hierarchical arrangement of schedulers is more effective (i.e., more

schedulers near the servers, near the root agent, or a balanced approach). For instance, this problem arose while doing a DIET deployment on a large Korean cluster, at the School of Aerospace and Mechanical Engineering (Seoul National University), which has more than 500 CPUs [1].

5.1 Deployment and Planning

A deployment is the mapping of a middleware platform across many resources. Deployment can be broadly divided in three categories: software deployment, system deployment, and an intermediate category using virtual machines:

Software deployment. maps and distributes a collection of software components on a set of resources, and can include activities such as configuring, installing, updating, and adapting a software system. Examples of tools that automate software deployment include SmartFrog [92], Distributed Ant [93], and Software Dock [96].

System deployment. involves assembling physical hardware as well as organizing and naming whole nodes and assigning activities such as master and slave. Examples of tools that facilitate this process include the Deployment Toolkit [156], Warewulf[1], and Kadeploy [122]. Although these toolkits can automate many of the tasks associated with deployment, they do not automate the decision process of finding an appropriate mapping of specialized middleware components to resources so that the best performance can be achieved from the system.

VM deployment. At an intermediary level, we find the deployment of Virtual Machines, which basically consists in deploying a system on top of another system. Several virtual machines can then be hosted by a single physical machine. This kind of deployment offers different levels of virtualization, as it can either just virtualize the system without hiding the hardware, or it can emulate another kind of hardware. We can cite Xen [24] and QEMU [26] as examples.

Our research focuses on the software deployment that provides an autonomic management. It can be of great help when managing a platform over a long period. Several solutions exist. They range from application specific to totally generic software.

ADAGE [116, 118] is a generic deployment software. It relies on modules specific for each software deployment. Its design decouples the description of the application from the description of the platform, and allows specific planning algorithms to be plugged-in. ADAGE is targeted towards static deployment (i.e., "one-shot" deployment with no further modifications), it can however be used in conjunction with CORDAGE [64] to add basic dynamic adaptations capabilities.

ADEM [101] is a generic deployment software, relying on Globus Toolkit. It aims at automatically installing and executing applications on the Open Science Grid (OSG), and can transparently retrieve platform description and information.

DeployWare [80, 81] is also a generic deployment software, it relies on the Fractal [137] component model. Mapping between the components and the machines has to be provided, as no automatic planning capabilities are offered.

Section 5.4 introduced GODIET [43]. It is a tool specifically designed to deploy the DIET middleware. An XML file containing the whole deployment has to be provided.

[1] http://www.warewulf-cluster.org

TUNe [35] is a generic autonomic deployment software. Like DeployWare, it also relies on the Fractal component model. TUNe targets autonomic management of software, i.e., dynamic adaptation depending on external events. Sekitei [107] is not exactly a "deployment software", as it only provides an artificial intelligence algorithm to solve the component placement problem. It is meant to be used as a mapping component for other deployment software. Finally, Weevil [170, 171] aims at automating experiment processes on distributed systems. It allows application deployment and workload generation for the experiments. However, it does not provide automatic mapping.

Apart from the installation and the execution of the software itself, which has been a well studied field, another important point is the **planning** of the deployment, i.e., the mapping between the software elements, and the computational resources. Whereas deployment software can cope with the installation and execution part, very few propose intelligent planning techniques. To the best of our knowledge, no deployment algorithm or model has been given for arranging the components of a Problem Solving environment (PSE) in such a way as to maximize the number of requests that can be treated in a time unit. In [117], software components based on the Corba component model are automatically deployed on the computational Grid. The Corba component model contains a deployment model that specifies how a particular component can be installed, configured and launched on a machine. The authors note a strong need for deployment planning algorithms, but to date they have focused on other aspects of the system. Our work is thus complementary.

Optimizing deployments is an evolving field. In [106], the authors propose an algorithm called Sekitei to address the Component Placement Problem (CPP). This work leverages existing AI planning techniques and the specific characteristics of CPP. In [108] the Sekitei approach is extended to allow optimization of resource consumption and consideration of plan costs. The Sekitei approach focuses on satisfying component constraints for effective placement, but does not consider detailed but sometimes important performance issues such as the effect of the number of connections on a component's performance.

The Pegasus System [154] workflow planning for the Grid as a planning problem. The approach is interesting for overall planning, when one can consider that individual elements can communicate with no performance impact. Our work is more narrowly focused on a specific style of assembly and interaction between components, and has a correspondingly more accurate view of performance to guide the deployment decision process.

5.2 Automatic Middleware Deployment Planning on Clusters

Joint work with:
⋆ Pushpinder Kaur Chouhan : ENS Lyon. LIP Laboratory. Lyon. France.
⋆ Frédéric Desprez : INRIA. LIP Laboratory. Lyon. France.

We have introduced an automated deployment planning approach that determines a good deployment for hierarchical scheduling systems in homogeneous cluster environments. We consider that a "good deployment" is one that maximizes the steady-state throughput of the system, i.e., the number of requests that can be scheduled, launched, and completed by the servers in a given time unit. We have shown that the optimal arrangement of agents is a complete spanning d-ary tree; this result is concordant with existing results in load-balancing and routing from scheduling and networking literature. More importantly, our approach automatically derives the optimal theoretical degree d for the tree. We developed the first detailed performance models available for scheduling and computation in the DIET system, and validated these models in a real-world

environment. We also presents real-world experiments demonstrating that the deployments automatically derived by our approach are in practice nearly optimal, and perform significantly better than other reasonable deployments.

In [44] we presented a heuristic approach for improving deployments of hierarchical NES systems in heterogeneous Grid environments. The approach is iterative; in each iteration, mathematical models are used to analyze the existing deployment, identify the primary bottleneck, and remove the bottleneck by adding resources in the appropriate area of the system. The techniques given in [44] are heuristic and iterative in nature and can only be used to improve the throughput of a deployment that has been defined by other means; the work presented hereafter provides an optimal solution to a more limited case, and does not require a predefined deployment as input.

5.2.1 Platform deployment

Our objective is to generate the best possible platform for the available resources, so as to maximize the throughput. The throughput is the number of requests that can be serviced for clients in a given time unit. We consider that at the time of deployment we do not know the clients' locations or the characteristics of the clients' resources. Thus, clients are not considered in the deployment process.

A valid deployment thus consists of a mapping of a hierarchical arrangement of agents and servers onto the set of resources. Any server or agent in a deployment must be connected to at least one other element; thus a deployment can only have connected nodes. A valid deployment will always include at least the root-level agent and one server. Each node can be assigned to either exactly one server, exactly one agent, or the node can be left idle.

Optimal deployment

Our objective is to find an optimal deployment of agents and servers for a set of resources. We consider an optimal deployment to be a deployment that provides the maximum throughput of completed requests per second. When the maximum throughput can be achieved by multiple distinct deployments, the preferred deployment is the one using the least resources.

We assume that at the time of deployment we do not know the locations of clients or the rate at which they will send requests. Thus it is impossible to generate an optimized, complete schedule. Instead, we seek a deployment that maximizes the steady-state throughput, i.e., the main goal is to characterize the average activities and capacities of each resource during each time unit.

Definition 1. *A **C**omplete **S**panning **d**-ary (CSD) tree is a tree that is both a complete d-ary tree and a spanning tree.*

For deployment, leaves are servers and all other nodes are agents. A degree d equal to one is useful only for a deployment of a single root agent and a single server. Note that for a set of resources and degree d, a large number of CSD trees can be constructed. However, with a given homogeneous resource set, all such CSD trees are equivalent as they provide exactly the same number of agents and servers, and thus provide exactly the same performance.

Definition 2. *A dMax set is the set of all trees for which the maximum degree is equal to dMax.*

Theorem 1. *In a dMax set, all dMax CSD trees have optimal throughput.*

Theorem 2. *A complete spanning d-ary tree with degree d that maximizes the minimum of the scheduling request and service request throughputs is an optimal deployment.*

5.2.2 A deployment model for DIET

This description of the agent/server architecture focuses on the common-case usage of DIET. An extension of this architecture with several hierarchies and several MA is also available (see Chapter 3).

Request performance modeling

We have designed a model for the scheduling throughput and the service throughput in DIET. We defined performance models to estimate the time required for various phases of request treatment in DIET. We made the following assumptions about DIET for performance modeling. The MA and LA are considered as having the same performance because their activities are almost identical and in practice we observe only negligible differences in their performance. We assumed that the work required for an agent to treat responses from SeD-type children and from agent-type children is the same. DIET allows configuration of the number of responses forwarded by agents; here we assume that only the best server is forwarded to the parent.

When client requests are sent to the agent hierarchy, DIET is optimized such that large data items like matrices are not included in the problem parameter descriptions (only their sizes are included). These large data items are included only in the final request for computation from client to server. As stated earlier, we assume that we do not have a priori knowledge of client locations and request submission patterns. Thus, we assume that needed data is already in place on the servers and we do not consider data transfer times.

The model is fully detailed in [62]. To give an overview, we can describe what we have modelized:

Agent communication model: To treat a request, an agent receives the request from its parent, sends the request to each of its children, receives a reply from each of its children, and sends one reply to its parent.

Server communication model: Servers only have one parent and no children, so we take into account the time in seconds required by a server for receiving messages associated with a scheduling request and the time in seconds required by a server for sending messages associated with a request to its parent.

Agent computation model: Agents perform two activities involving computation: the processing of incoming requests and treatment of replies. There are two activities in the treatment of replies: a fixed cost in MFlops and a cost that is the amount of computation in MFlops needed to process the server replies, sort them, and select the best server.

Server computation model: Servers also perform two activities involving computation: performance prediction as part of the scheduling phase and provision of application services as part of the service phase.

Steady-state throughput modeling

We have considered two different theoretical models for the capability of a computing resource to do computation and communication in parallel:

Send or receive or compute, single port: In this model, a computing resource has no capability for parallelism: it can either send a message, receive a message, or compute. Only a single port is assumed: messages must be sent serially and received serially. This model may be reasonable for systems with small messages as these messages are often quite CPU intensive.

Send || receive || compute, single port: In this model, it is assumed that a computing resource can send messages, receive messages, and do computation in parallel. We still only assume a single port-level: messages must be sent serially and they must be received serially.

Based on all these considerations we compared our model with different real deployments on Grid'5000 [31]. Figure 5.1 gives an example of the relevance of the model. We can see the comparison between the model and the measured throughput when we deployed 45 nodes providing the matrix multiplication (`dgemm` function from BLAS library [12]).

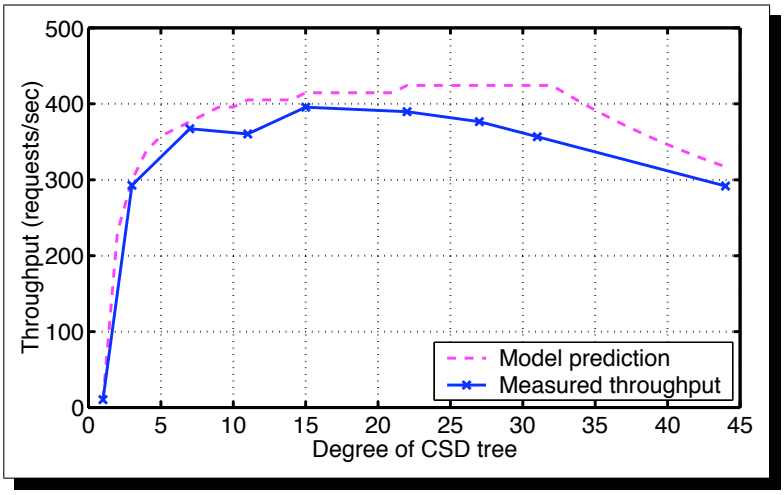


Figure 5.1: Predicted and measured throughput for different CSD trees for DGEMM 310 with 45 available nodes in the Sophia cluster.

5.3 Automatic Middleware Deployment Planning on Heterogeneous Environment

Joint work with:
- ⋆ Benjamin Depardon : ENS Lyon. LIP Laboratory
- ⋆ Frédéric Desprez : INRIA. LIP Laboratory

The previous model gives a solution for hierarchical middleware, and algorithms to deploy a hierarchy of schedulers on cluster and Grid environments. We can notice that we obtain an optimal hierarchical middleware deployment for a homogeneous resource platform of a given size.

The maximum throughput of the middleware is modelized when it works in steady-state, meaning that only the period when the middleware is fully loaded is taken into account, and not the periods when the workload initially increases, or decreases in the end. However, a limitation in this

latter work is that only one kind of service could be deployed in the hierarchy. Such a constraint is of course not desirable, as nowadays many applications rely on workflows of different services. Hence, the need to extend the previous models and algorithms to cope with hierarchies supporting several services.

Several works have also been conducted on Corba-based systems. Works on throughput, latency and scalability of Corba have been conducted [89, 90], or even on the ORB architecture [7]. Several Corba benchmarking frameworks have been proposed [36], and some web sites also propose benchmarks results, see for example [138]. These benchmarks provide low level metrics on Corba implementations, such as consumed memory for each data type, CPU usage for each method call. Though accurate, these metrics do not fit in our study, which aims at obtaining a model of the behavior of DIET at a user level, and not at the methods execution level.

Two techniques are used to solve the deployment planning problems. The two methods show quite orthogonal ways of thinking:

Linear Programming (LP) aims at obtaining a solution to an optimization problem through exact resolution of a system of constraints. LP is a method for the optimization of a linear function (the objective function), subject to several constraints which can be represented as linear equalities or inequalities. There exists several software to solve LP problems, such as GNU Linear Programming Kit (GLPK) [4], ILOG CPLEX [5], or lpsolve [6].

Genetic Algorithms try to find a good solution through random searches and improvements. Genetic Algorithms [126] are part of what are called meta-heuristics such as tabu search, hill climbing, or simulated annealing. More precisely genetic algorithms are evolutionary heuristics. Genetic algorithms were invented by John Holland in the 1970s [100], and are inspired by Darwin's theory about evolution. The idea is to have a population of abstract representations (called chromosomes or the genotype of the genome) of candidate solutions (called individuals) to an optimization problem evolve towards better solutions. There exists quite a lot of parallel distributed evolutionary algorithms and local search frameworks, such as DREAM [18], MAFRA [115], MALLBA [9], and ParadisEO [37].

5.3.1 Planning with heterogeneous services

We concentrate on determining what is the best mapping of the middleware on the available resources to fulfill users' requirements. Platforms can also be of different "flavors": homogeneous or heterogeneous. All in all, four problems are addressed during this work (i.e., taken into account many services).

1. **Modelization.** We provide a model for hierarchical GridRPC middleware. It is an extension of the one presented ahead and in [62], in that it takes into account several services.

2. **Homogeneous platform.** This is the simplest platform we can deal with: all nodes have the same characteristics, and are interconnected with links which also have the same characteristics. This model reflects the common case of **cluster computing**.

3. **Computation heterogeneous platform.** Going a step further in this model, we consider that nodes can have different characteristics, even though the interconnection links remain homogeneous. This reflects the case where several machines share the same network: for example clusters that are linked altogether, or company desktop computers.

4. **Heterogeneous platform.** The last step is of course the case where every machine, and every link can have its own characteristics. This is the typical Grid computing case.

The previous approach focus on one kind of service. To extend the model we added the scheduling throughput for different service offered by the platform, i.e., the rate at each request are processed by the scheduling phase. We have shown in [62] that the completed request throughput of a deployment is given by the minimum of the scheduling and the service request throughput. Moreover the service request throughput for a given service increases as the number of servers included in a deployment and allocated to this service increases.

First we have designed a model for homogeneous platform [46]. From this model we have proposed a heuristic for automatic deployment planning. The heuristic comprises two phases. The first step consists in dividing nodes between the services, so as to support the servers. The second step consists in trying to build a hierarchy, with the remaining nodes, which is able to support the throughput generated by the servers.

First we need to allocate server nodes, then a bottom-up approach is used to build a hierarchy of agents. The aim is to obtain for all services the same ratio of requested to obtained throughput. A simple way of dividing the available nodes to the different services is to increase iteratively the number of assigned nodes per services.

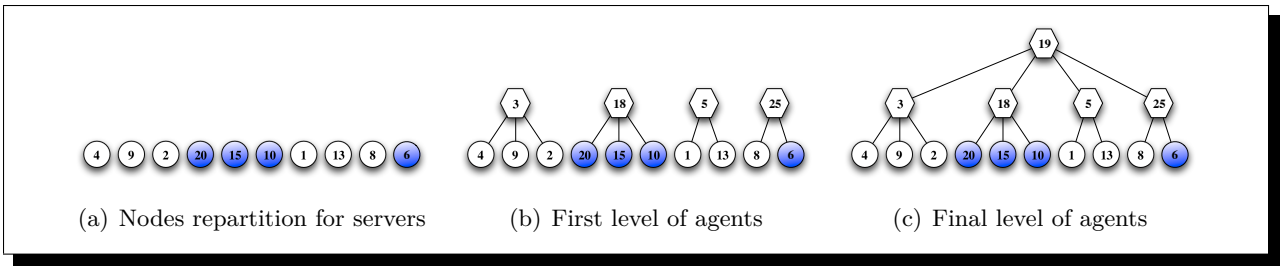
As previously, we keep the bottom-up construction. We first distribute some nodes to the servers. Then, with the remaining nodes, we iteratively build levels of agents. Each level of agents has to be able to support the load incurred by the underlying level. The construction stops when only one agent is enough to support all the children of the previous level. In order to build each level, we make use of an integer linear program fully described in [73]. Using the linear program, we can recursively define the hierarchy of agents, starting from the bottom of the hierarchy. Our objective function is the minimization of the number of agents: the equal share of obtained throughput to requested throughput ratio has already been cared of when allocating the nodes to the servers, hence our second objective that is the minimization of the number of agents in the hierarchy has to be taken into account.

To build the hierarchy we proceed as follow. We first try to give as many nodes as possible to the servers, and we try to build a hierarchy on top of those servers with the remaining nodes. Whenever building a hierarchy fails, we reduce the number of available nodes for the servers. Hierarchy construction may fail for several reasons

- no more nodes are available for the agents
- no solution
- only chains of agents have been built, i.e., each new agent has only one child. However, chains of agents in a hierarchy is useless.

Finally, either we return a hierarchy if we found one, or we return a hierarchy with only one child of each type, as this means that the limiting factor is the hierarchy of agents. Thus, only one server of each type of service is enough, and we cannot do better than having only one agent. Figure 5.2 presents an example of our bottom up approach.

Correcting the throughput. Once the hierarchy has been computed, we need to correct the throughput for services that were limited by the agents. Indeed, the throughput computed may



(a) Nodes repartition for servers (b) First level of agents (c) Final level of agents

Figure 5.2: Bottom-up approach.

be too restrictive for some services. The values obtained implied that we had effectively an equal ratio between obtained throughput over requested throughput for all services, which may not be the case if a service requiring lots of computation is deployed alongside a service requiring very few computation. Hence, once the hierarchy is created, we need to compute what is really the throughput that can be obtained for each service on the hierarchy. To do so, we simply use our agent model, and we try to maximize the values of the throughput for all services that are limited by the agents. A linear program was written for this purpose fully described in [73].

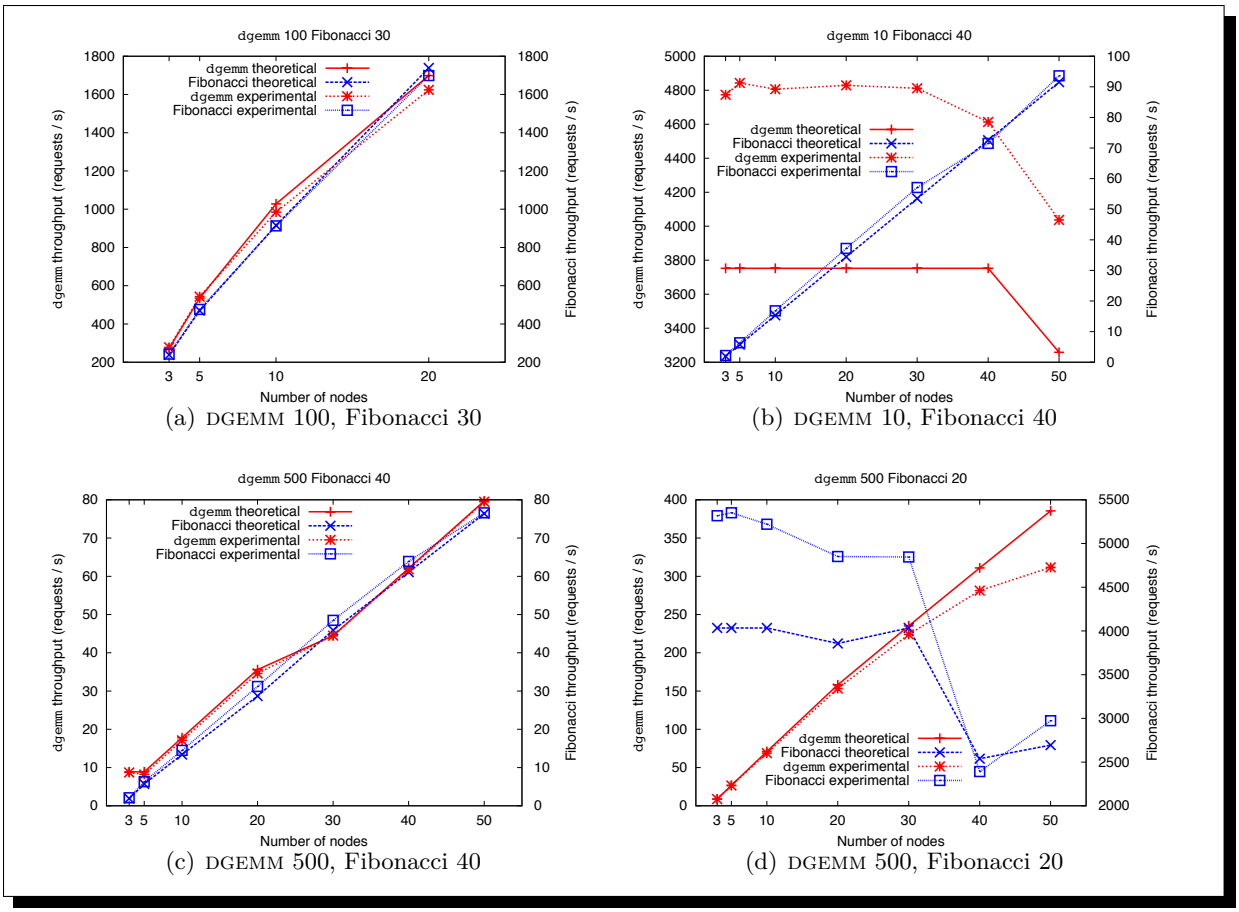
Comparing DIET and the Model We confronted our model with our target middleware, DIET, over the Grid'5000 platform. We will denote by DGEMM x the call to a DGEMM service on matrices of size $x \times x$, and Fibonacci x the call to a Fibonacci service to compute the Fibonacci number for $n = x$. We have considered two computation/communication (as mentioned in Section 5.2.2 page 56)

To validate the model we have benchmarked the communication and the computation performance of the platform, and the communication and computation requirements of the DIET middleware [73]. Then, we compared throught given by the model and the throughput obtained with a real experiment.

Experimental Results. Table 5.1 presents the relative error between the theoretical and experimental throughput: a dash in the table means that the algorithm returned the same hierarchy as with fewer nodes, and hence the results are the same. Figures 5.3(a), 5.3(b), 5.3(c) and 5.3(d) compare the throughput obtained with our model and during our experiments.

As can be seen, the experimental results closely follow what the model has predicted. Higher errors can be observed for really small services (DGEMM 10 and Fibonacci 20). This is due to the fact that really small services are harder to benchmark. Note however that even if theoretical throughputs for small services are lower than experimental ones, the model correctly detects when the throughput drops due to the load at the agent level (see Figures 5.3(b) and 5.3(d)). Figure 5.3(a) does not extend to 50 nodes, this is due to the fact that our algorithm returned exactly the same hierarchy for more than 20 nodes. Adding new servers to this hierarchy did not improve the performance.

In Figure 5.3(d), we can observe that DGEMM performance degrades as the number of nodes increases. This seems to come from Corba communications. The modification of Corba can increased greatly the obtained throughput. However, we did not manage to make the experiments exactly match the model with DGEMM 500, Fibonacci 20 experiments.



(a) DGEMM 100, Fibonacci 30

(b) DGEMM 10, Fibonacci 40

(c) DGEMM 500, Fibonacci 40

(d) DGEMM 500, Fibonacci 20

Figure 5.3: Homogeneous: comparison theoretical/experimental throughput.

Experiment		**Number of nodes**						
		3	**5**	**10**	**20**	**30**	**40**	**50**
DGEMM 100, Fibonacci 30	DGEMM	1.87%	1.58%	4.17%	4.39%	-	-	-
	Fibonacci	1.76%	1.26%	0.26%	2.28%	-	-	-
DGEMM 10, Fibonacci 20	DGEMM	27.4%	-	-	-	-	-	-
	Fibonacci	27.3%	-	-	-	-	-	-
DGEMM 10, Fibonacci 40	DGEMM	27.2%	29.1%	28.1%	28.7%	28.2%	22.9%	23.9%
	Fibonacci	14.6%	10.4%	9.7%	8.0%	6.7%	1.4%	2.3%
DGEMM 500, Fibonacci 20	DGEMM	1.1%	0.7%	3.0%	3.0%	4.7%	9.5%	19.1%
	Fibonacci	31.8%	32.6%	26.4%	25.8%	20.1%	5.8%	10.4%
DGEMM 500, Fibonacci 40	DGEMM	2.2%	7.9%	3.9%	2.8%	0.0%	0.5%	0.3%
	Fibonacci	10.5%	10.5%	8.2%	8.7%	5.7%	4.6%	0.3%

Table 5.1: Experimental throughput: relative error.

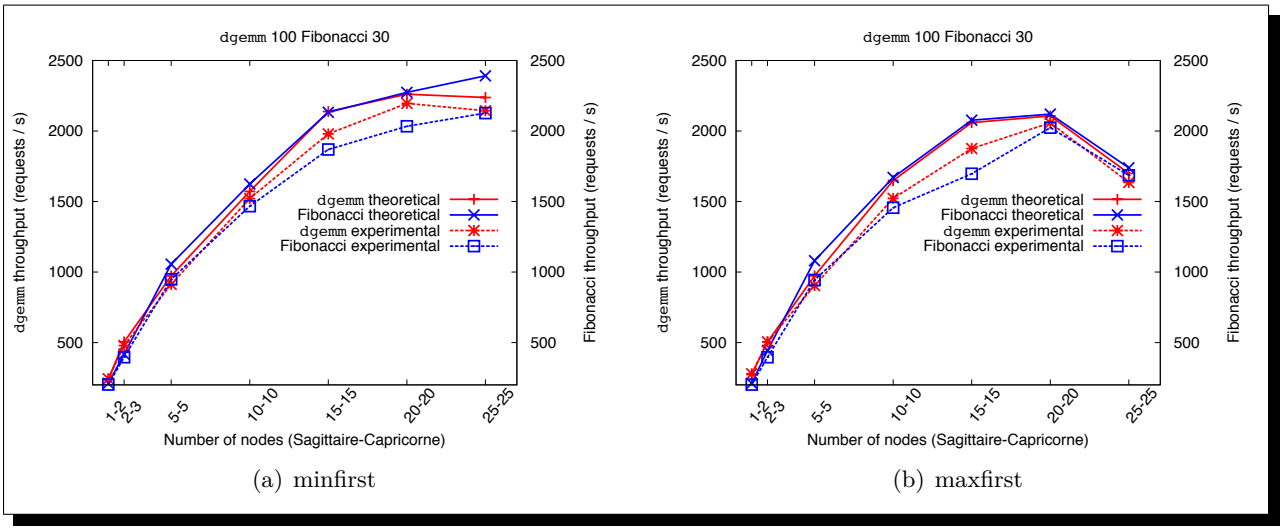
In order to validate the relevancy of our algorithm to create hierarchies, we compared the throughputs obtained with our hierarchies, and the ones obtained with a star graph having exactly the same allocation of servers obtained with our algorithm. Nevertheless our algorithm gives better throughputs than star-graphs thus validating the benefit of our approach.

5.3.2 Planning with heterogeneous computation

Before moving to a fully heterogeneous platform, we first deal with a simpler case. We assume that nodes' computing power can be heterogeneous, but that communication links are homogeneous with fixed bandwidth. This type of platform is quite close to a fully homogeneous one, and thus does not discard our bottom-up approach presented in Section 5.2. In fact, we only need to adapt the linear program and algorithm based on to take into account the computing power heterogeneity.

We also use the same approach to distribute the nodes for the servers. Thus, we reused our algorithm with the only modification being that we need to take into account the nodes' heterogeneity. Hence, we propose two heuristics: min-first which first give the less powerful nodes to the servers, and max-first which first give the more powerful nodes to the servers. As previously, algorithm first computes the maximum supported throughput by an agent (we use for this the most powerful node), then distributes some nodes to the servers using either min-first or max-first heuristics, then tries to build a hierarchy of agents using the linear program. Once a hierarchy is found, we correct the obtained throughput.

As has been done for the homogeneous platform case, we validate our model against real executions with the DIET middleware. Figures 5.4(a) and 5.4(b) present the comparison between theoretical and experimental throughput for respectively experiments with min-first and max-first heuristics. Table 5.2 presents the relative error for those experiments. As can be seen, the min-first heuristic is the most interesting one when dealing with large platforms. This can easily be explained. Using less powerful nodes for servers leaves more powerful nodes for the agents, hence the maximum attainable throughput due to agents limitation is higher. Whatever the heuristic, we remain within 18.3% of the theoretical model.



(a) minfirst (b) maxfirst

Figure 5.4: DGEMM 100, Fibonacci 30 theoretical and experimental throughput, with min-first and max-first heuristics.

Experiment		Number of nodes						
		1-2	**2-3**	**5-5**	**10-10**	**15-15**	**20-20**	**25-25**
min-first	DGEMM	0.3%	5.2%	6.1%	3.2%	3.9%	2.9%	4.20%
	Fibonacci	4.1%	5.1%	10.2%	9.7%	10.0%	10.6%	11.0%
max-first	DGEMM	2.1%	0.1%	7.0%	7.7%	8.9%	2.3%	3.8%
	Fibonacci	4.6%	10.8%	12.7%	12.8%	18.3%	4.5%	3.0%

Table 5.2: Relative error, using min-first and max-first heuristics.

Hierarchy shape Figures 5.5, and 5.6 give an example of the shape of the hierarchy generated by respectively min-first and max-first on a 25-25 platform. Colored nodes are on the first cluster (Sagittaire on Grid'5000), white nodes on the second cluster (Capricorne on Grid'5000) cluster. "D" stands for DGEMM, and "F" stands for Fibonacci.

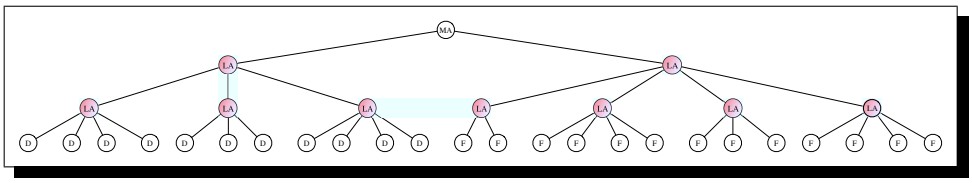


Figure 5.5: Hierarchy generated with min-first on 25-25 nodes.

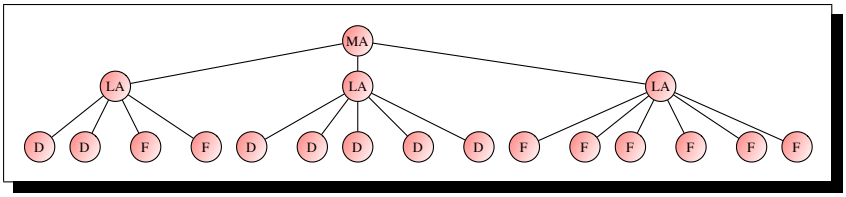


Figure 5.6: Hierarchy generated with max-first on 25-25 nodes.

5.3.3 Planning with heterogeneous computation and communication

We finally deal with a fully heterogeneous platform: each node has a computing power of its own, and the communication links between any two nodes are possibly all different.

The problem when dealing with totally heterogeneous platforms is that we need information about both the location of the parent, and location of the children. Hence, the bottom-up approach we used so far will not be applicable, nor would be a top-down approach: we will not be able to build a level if we do not know the position of the parent in a bottom-up approach, and conversely we cannot build a level without knowing the position of the children in a top-down approach.

Genetic Algorithm Approach

As we cannot use an iterative approach to build a hierarchy without risking to have to test all possible solutions, we took a totally different approach: we rely on a **genetic algorithm** to generate a set of hierarchies, then evolve them, and finally select the best one among them.

In order to define our genetic algorithm, we need to describe a few notions: the objective function, crossover and mutations, and finally the evaluation strategy.

The objective function has to encode all the goals we aim at optimizing in a hierarchy. It also needs to be subject to an order relation. The chosen fitness value encodes the fact that we want to maximize the throughput, while minimizing the number of agents. Actually, in order to guide a bit more the genetic algorithm towards convergence, we encodes two more metrics that we wish to minimize at the end of the fitness value: the number of agents that do not have any children (in order to remove really useless elements) and the depth of the hierarchy (this should not affect the throughput, but this impacts the response time of the hierarchy, and limits the formation of chains of agents). The genotype needs to encode the whole hierarchy: the parent/children relationship, and the type of each node (agent, server or unused). Two arrays are used for this. Genotypes are randomly generated when creating the first generation of individuals: nodes' types and relationship are randomly chosen in such a way that a valid hierarchy is created.

We define a crossover between two hierarchies as follows. Crossovers are only made on the parent array. We randomly select two nodes (one on each hierarchy) and exchange the parent of both selected nodes. Figure 5.7 presents an example of crossover. Why not define a crossover which replaces a whole part of a hierarchy into another one? This approach works well for a small number of nodes, but it has a far too big impact on the hierarchy shape on a large number of nodes.

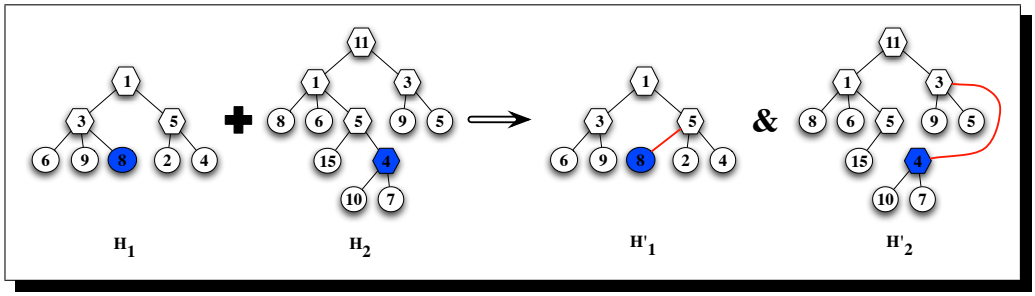


Figure 5.7: Crossover. Colored nodes are the one selected for crossover, within hierarchy elements represent the nodes' number.

Mutation

Mutations on a hierarchy can occur at different levels in the hierarchy. We define the following mutations, also presented in Figure 5.8:

- Hierarchy modification:

    1. we randomly select a node to change its type. If the mutation changes the type from agent to unused or SeD, or from SeD to unused, then we remove the underlying hierarchy

and modify the type of the node. If the type changes from unused to agent or SeD, we randomly choose a parent among the available agents.

2. we randomly select a node that will choose a new parent among the available agents. We can end up with two hierarchies (if the new parent is the node itself). In this case we randomly select one of the two hierarchies, and delete the other.

- Pruning: a node is randomly selected, then its whole underlying hierarchy is deleted.

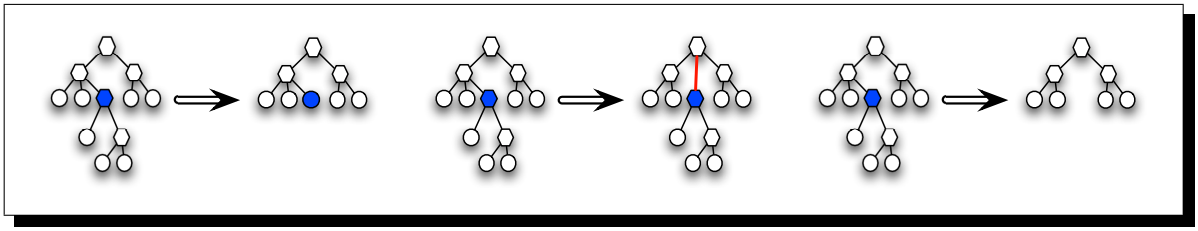


Figure 5.8: Mutations. Colored node has been selected for mutation. Hexagons are agents, and circles are servers.

Planing evaluation

In order to evaluate a hierarchy generated by our genetic algorithm, we first compute for all services the throughput supported by the servers. Then, for each agent in the hierarchy, we compute the maximum throughput for each service supported by the agent.

Genetic Algorithms (GA) rely on quite a lot of different parameters. Each one of them can influence the quality of the result. Among them are the following: Selection method (we need to choose which one should stay alive), weak elitism (forces the algorithm to converge towards a locally good solution), size of the population, Probability of crossover and mutation. We used the ParadisEO [37] framework to implement our genetic algorithm.

We have compared the best GA to the min-first and max-first heuristics. We observed that even if on the mean GA do not obtain results as good as min-first or max-first, the best GA results closely follows the min-first heuristic. The loss is no bigger than 15%, and it gives better results than max-first for larger platforms. This confirms that our approach can be effective: even if one run of GA is not sufficient to obtain the best result, taking the best hierarchy over a few runs of GA can give us good results. Note that this is often the case with genetic algorithms, as it is an exploratory method. We are quite confident that the performance loss obtained with the GA solutions can be reduced by fine tuning the GA parameters.

5.4 GoDiet

Joint work with:

⋆ Holly Dail : INRIA. LIP Laboratory. Lyon. France.

We have introduced GoDiet the software deployment tool of Diet in the Section 2.8.1. The goal of GoDiet is to automate the deployment of Diet platforms and associated services for diverse Grid environments. As we mentioned users write this XML file, describing their available compute and storage resources and the desired overlay of agents and servers onto those resources. Users can use the previous work concerning the planning to find the best deployment for a given platform.

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 [32]; we have chosen Java for the 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 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, 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. Thus it's easy to design a tool to generate automatically the XML file using the deployment algorithm based on the planning model.

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 mode batch where the platform can be launched and stopped without user interaction; this mode is primarily useful for experiments.

We give here some details around the internal part of GoDiet to help the reader to have a good overview of this tool. We use `scp` and `ssh` to provide secure file transfer and task execution. `ssh` is a tool for remote machine access that has become almost universally available on Grid resources in recent years. With a carefully configured `ssh` command, 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 then 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.

GoDiet was used to perform many large scale deployment [22, 38, 39, 43]. One of future work is to provide a solution with the complete deployment process. Means integrate the planning algorithms into GoDiet.

5.5 Conclusion

In this chapter, we have presented our approach for designing a model for hierarchical middleware performance. Several parameter have to be taken into account: homogeneous platform, heterogeneous services, heterogeneous computation, heterogeneous computation and communication. We propose different ways to solve the problem using the model: linear programming paradigm and genetic algorithm approach. To offer a complete and automatic deployment tool a huge problem is not yet solved and concern the resources discovery. It is very difficult to have an automatic resources discovery on a Grid environment. Nevertheless from the knowledge of resources, the planing solutions given in this chapter and the usage of GoDiet offer a powerful toolkit to design an efficient deployment. The next work will be to work on a dynamic deployment and update the deployment according to the usage of the middleware.

Chapter 6

Data management in DIET

Contents

Data management in grid environments is currently a topic of major interest to the grid computing community. Indeed, in many application, high performance computing requires to deal with a large amount of data. Sometimes the large data comes from the input data, or it is generated during runtime, or it concerns the output data, and of course all combinations of these three cases exists. However, when we started this work no approach had been widely established for transparent data sharing on grid infrastructures. Currently, the most widely-used approach for data management for distributed grid computation relies on explicit data transfers between clients, and computing servers: the client has to specify where the input data is located, and to which server it has to be transferred. Then, at the end of the computation, the results are eventually transferred back to the client. As an example, the Globus [82] platform provides data access mechanisms based on the GridFTP protocol [10]. Though this protocol provides authentication, parallel transfers, checkpoint/restart mechanisms, etc., it still requires explicit data localization. It has been shown that providing data with some degree of persistence may considerably improve the performance of series of successive computations. In another direction, a large-scale data storage system is provided by IBP [5], as a set of so-called buffers distributed over Internet. The user can "rent" these storage areas, and use them as temporary buffers for optimizing data transfers across a wide-area network. Transfer management still remains the burden of the user. Finally, Stork [114] is another

example of system providing mechanisms to explicitly locate, move and replicate data according to the needs of a sequence of computations. It provides the user with an integrated interface to schedule data movement actions just like computational jobs. Again, data location and transfer have to be explicitly handled by the user. In this Chapter we will see how the data management is taken into account in the GridRPC paradigm. And we describe three data management systems involved within DIET.

6.1 GridRPC and Data Management

The GridRPC paradigm is based on the classical RPC model to allow developers and users to transparently take benefits of the Grid resources. In such a platform, servers declare one or several services which are accessible to the client through a simple function call. The GridRPC middleware have to manage the services retrieval, the transfers of the parameters and task's execution on the server before to send back the results to the client. Everything should be done as transparently as possible, masking the Grid complexity to the users and application developers.

However, in a large scale heterogeneous platform, the data transfer time can be very long, and the middleware have to provide mechanisms avoiding useless transfers or data movements. Some strategies have been studied to avoid such transfers:

Data persistency: Many applications use some temporary or intermediate results. For example, the matrices operation $A \times B + C$, needs to calculate first $D \leftarrow A \times B$ then $D + C$. The intermediate result D can be uninteresting for the user. In another hand, the matrices multiplication operation can be used many times by different applications, and should be proposed independently. A way to avoid the intermediate result D to go back to the client application, which have then to send it again, is to declare data D has a *persistent* result. That means that this data must stay on the server where it was computed. Then, just giving a reference to this data, the client can use it for the rest of the treatment. For workflows, and dataflows executions the exception becomes the rule, and many intermediate results should not be transferred to the client application to avoid bandwidth, and time consumption.

Data locality: Some scientific applications use very large data or data sets which are not directly interesting for the users. For example, when proceeding to a BLAST alignment search on large biological database [12, 49], the most important scientific aspect is the result of the BLAST search on the database and not the database itself. The researchers do not need to store the database on their local machine. It is often the case in biology, high energy physics, cosmology etc. Similarly, in order to preserve storage and to avoid large data transfer between a personal computer using a small bandwidth connexion, and the large bandwidth network of a Grid platform, the Grid middleware should be able to store data, and to provide an easy access to them. Moreover, to increase the platform performance, and to preserve network bandwidth, the data should be stored as "near" as possible to the computation resources which use them.

Data replication: The data involved in scientific in-silico researches are often used in an embarrassingly parallel manner. Indeed, as reference data, they are explored, compared to new ones or analysed using many different parameters. Each search on them can then be done independently as long as the computation resources can access to the reference data. Staying on the same example, the BLAST application uses reference biological databases to search alignment

between the sequences they contain, and newly discovered sequences. The main objective is to find clues about the function of an unknown gene or protein. Several thousands of new sequences are usually compared to one or several biological databanks. In such a context, disposing of several data replica is mandatory to efficiently parallelize the searches.

These three strategies are tightly linked. Indeed, data replication, and data locality implies data persistency; to be efficient the data replication must take into account the proximity of the computing resources; to allow more complex workflows than only iterative processes, we need data persistency as well as data replication.

In the GridRPC context, the computing resources are the services, and the data are the input parameters, and the results of these services. To provide these data management strategies, a GridRPC middleware have to implement:

- Different data persistency modes.
- An explicit data placement mechanism.
- A data replication system.

The three next sections present these needs in details.

6.1.1 Data persistency mode

As we have seen before, leaving the data on the server that produces them can avoid useless transfers. It is particularly the case for intermediate or temporary data. However, sometimes, these intermediate results can also be interesting for the user. In these cases, the data should stay on the server (to avoid the client application has to send them again on the platform), but it also has to be returned to the client. We can define two data persistency modes:

- *Persistent data:* The data stays where it was produced.
- *Persistent-return data:* The data stays where it was produced and is also returned to the client.

It is also possible that a user wants that a data stays on a server for as long as he wants, until he decides to remove it. For example, the user knows that the data will be frequently used as a service parameter. We can define a persistency mode which gives a *data lifetime* information. As for the persistent data mode, the user can also want to get the intermediate results back. We can define two other persistency modes:

- *Sticky data:* The data stays where it was produced, until the user explicitly chooses to remove it.
- *Sticky-return data:* The same than the sticky mode, but the data is also returned back to the client application.

The last persistency mode for a data is no persistency at all, the data is returned back to the client, and immediately removed from the server. We will call such a data "*Volatile data*".

Introducing persistent data, the middleware must provide data naming, and retrievement services. Indeed, to reuse a data which is stored on the Grid, we have to give a *unique reference* to it (a data unique identifier), and the middleware must be able to retrieve the data using only this identifier.

6.1.2 Data placement

To move the data closer to the computing services which use them, the GridRPC middleware can either allow the user to choose explicitly the server or select the server itself. In the first case, the user has to know more information about the platform, but can also do more optimization taking into account his data usage. By letting the middleware choose where to store the data, the user must give more information about the applications, and the data they could use. In both cases, the middleware must be able to move the data from a server to another one. So, it needs a data transfer service which can retrieve, and move the data. Because the storage resources are limited, the middleware should also be able to select which data to remove from a server when it is necessary to store a new one. In this situation, the data *sticky* mode is relevant. Indeed, by choosing the sticky mode for a data, the middleware avoids to remove it to free some space to a new one.

The middleware then should implement a data replacement algorithm to select the data which can be removed. Several strategies can be implemented like the *Least Frequently Used* or *Last Recently Used* data selection algorithms.

6.1.3 Data replication

Data replication allows to use the same data as entry for several services or for the same service on different servers. It is then possible to fully parallelize the embarrassingly parallel or parameter sweep applications. Data replication can also preserve the platform bandwidth, avoiding long distance transfers, and increases the data availability. However, data replication introduces several issues:

- When a data have several replicas, which one should be used for a particular transfer ? (best "data source" selection)
- When modifying a data, how and when its replicas should be updated ? (data coherency problem)
- When removing a data replica, how can we ensure that at least one copy of it stays on the platform ? (data availability problem)

There are two ways to manage data replication:

Explicit data replication: The user chooses explicitly where and when to replicate a data. For example, he can choose to replicate a data on a particular server, or as much as possible on the entire platform.

Implicit data replication: Instead of simply moving a data from one server to another one, the middleware can choose to copy it on the second one, leaving a copy on the first one. Using a service with a data as entry parameter causes this data replication.

The data replacement algorithm involved in the data placement service is an important point when using data replication. Selecting a good strategy can save network bandwidth as well as increase the platform performance. Choosing the data persistency mode for each data replica (simply persistent or sticky) is then also important for the platform performance, and data availability.

6.1.4 GridRPC data management API

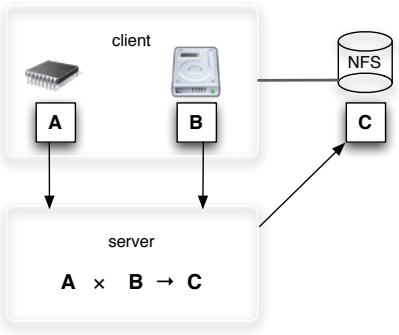
Joint work with:

⋆ Yves Caniou	: University Claude Bernard 1. LIP Laboratory. Lyon. France.
⋆ Gaël Le Mahec	: ENS-Lyon. LIP Laboratory. Lyon. France.
⋆ Hidemoto Nakada	: National Institute of Advanced Science and Technology. Tokyo. Japan.
⋆ Yoshio Tanaka	: National Institute of Advanced Science and Technology. Tokyo. Japan.

A good way to understand and show the functionalities of the GridRPC API is to describe some examples.

Simple data management using the GridRPC API

In the example given Figure 6.1, the client proceeds to a simple remote call using a memory data as first parameter, and a disk data as second parameter for the call. It defines the result to be written on an NFS partition on its local network.




```
grpc_function_handle_init(hdl, "server1.somewhere.eu", "x");
grpc_data_memory_mapping_set("A", &A);
grpc_data_init(data1, {"memory://local.here.fr/&A", NULL}, {NULL},
               GRPC_DOUBLE, {n, m, NULL}, {GRPC_VOLATILE, NULL});

grpc_data_init(data2, {"file://home/me/matrix.dat", NULL}, {NULL},
               GRPC_DOUBLE, {m, n, NULL}, {GRPC_VOLATILE, NULL});

grpc_data_init(data3, {NULL}, {"nfs://local.here.fr/home/me/C.dat",
               NULL}, GRPC_DOUBLE, NULL, NULL);

grpc_call(hdl, &data1, &data2, &data3);
```

Figure 6.1: GridRPC simple data management

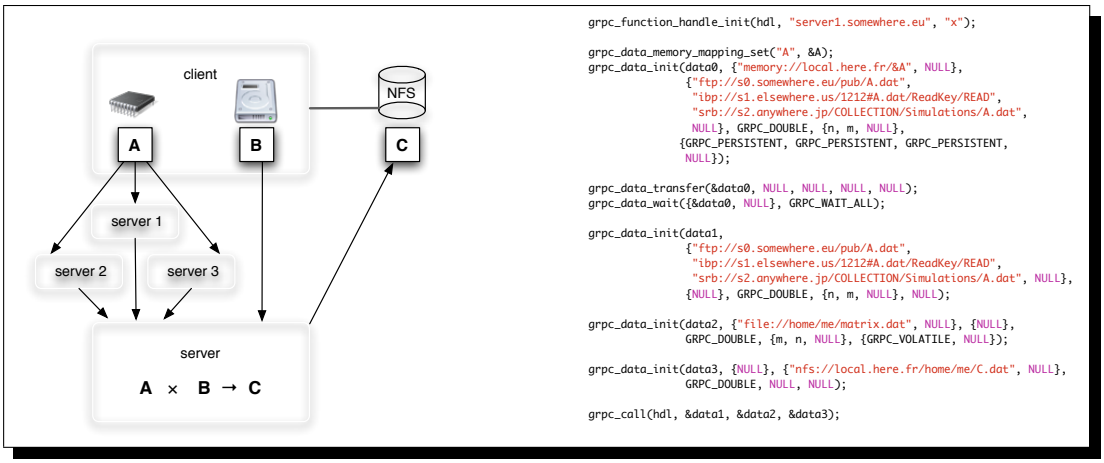
Data replication using the GridRPC API

In Figure 6.2, the client proceeds to the data replication, and then uses these different replicas as input URI for the first parameter. The result is copied on several remote servers, and stays in place for a second service call.

Data prefetching using the GridRPC API

The example describes in Figure 6.3, the client starts to send the data to one server while using another one for a service execution. The service sends its result to the second server, and the client calls a new service on it using the prefetched data, and the first service result as parameters.

During the history of DIET, three data managers were designed. With different approaches and different goals. The first data manager available in DIET is called DTM [71], the second one is based on a P2P approach, and finally DAGDA, the third one, that offers a complete implementation of the GridRPC data management API.




```
grpc_function_handle_init(hdl, "server1.somewhere.eu", "x");

grpc_data_memory_mapping_set("A", &A);
grpc_data_init(data0, {"memory://local.here.fr/&A", NULL},
               {"ftp://s0.somewhere.eu/pub/A.dat",
                "ibp://s1.elsewhere.us/1212#A.dat/ReadKey/READ",
                "srb://s2.anywhere.jp/COLLECTION/Simulations/A.dat",
                NULL}, GRPC_DOUBLE, {n, m, NULL},
              {GRPC_PERSISTENT, GRPC_PERSISTENT, GRPC_PERSISTENT,
               NULL});

grpc_data_transfer(&data0, NULL, NULL, NULL, NULL);
grpc_data_wait({&data0, NULL}, GRPC_WAIT_ALL);

grpc_data_init(data1,
               {"ftp://s0.somewhere.eu/pub/A.dat",
                "ibp://s1.elsewhere.us/1212#A.dat/ReadKey/READ",
                "srb://s2.anywhere.jp/COLLECTION/Simulations/A.dat", NULL},
               {NULL}, GRPC_DOUBLE, {n, m, NULL}, NULL);

grpc_data_init(data2, {"file://home/me/matrix.dat", NULL}, {NULL},
               GRPC_DOUBLE, {m, n, NULL}, {GRPC_VOLATILE, NULL});

grpc_data_init(data3, {NULL}, {"nfs://local.here.fr/home/me/C.dat", NULL},
               GRPC_DOUBLE, NULL, NULL);

grpc_call(hdl, &data1, &data2, &data3);
```

Figure 6.2: Data replication example with the GridRPC API


```
grpc_function_handle_init(hdl1, "server1.somewhere.eu", "x");
grpc_function_handle_init(hdl2, "server2.elsewhere.jp", "+");

grpc_data_memory_mapping_set("A", &A);
grpc_data_init(data1, {"memory://local.here.fr/&A", NULL},
               {"ftp://server2.elsewhere.jp/pub/A.dat", NULL},
               GRPC_DOUBLE, {n, m, NULL}, {GRPC_VOLATILE, GRPC_PERSISTENT});
grpc_data_transfer(&data1, NULL, NULL, NULL);

grpc_data_init(data2, {"file://home/me/matrix.dat", NULL}, {NULL},
               GRPC_DOUBLE, {m, n, NULL}, {GRPC_VOLATILE, NULL});
grpc_data_init(data3, {NULL}, {"nfs://server2.elsewhere.jp/home/me/C.dat", NULL},
               GRPC_DOUBLE, NULL, NULL);
grpc_data_init(data4, {NULL}, {"nfs://local.here.fr/home/me/D.dat", NULL},
               GRPC_DOUBLE, NULL, NULL);

grpc_call(hdl1, &data1, &data2, &data3);
grpc_call(hdl1, &data1, &data3, &data4);
```

Figure 6.3: Data prefetching example with the GridRPC API.

6.2 DTM: Data Tree Manager

Joint work with:
- ⋆ Bruno Del Fabro : University of Franche-Compté. LIFC Laboratory. Besançon. France.
- ⋆ Frédéric Desprez : INRIA. LIP Laboratory. Lyon. France.
- ⋆ Jean-Marc Nicod : University of Franche-Comté. LIFC Laboratory. Besançon. France.
- ⋆ Laurent Philippe : University of Franche-Comté. LIFC Laboratory. Besançon. France.

DTM is the data manager used by DIET for the versions earlier than 2.3. DTM provides a data management which allows the users to define a data as persistent to avoid useless transfers. DTM is composed of two main components:

- The *Data Location Manager* which is used to retrieve a data stored on the DIET hierarchy. Each element of the DIET hierarchy has its own data location manager.
- The *Data Manager* which is only launched on the SeDs. It is used to store, delete, and record the data on a node.

The Data Location Manager fills and updates a data location table at each level of the hierarchy. For each data identifier, it records on these tables which child of the current node should be followed to retrieve the data. This mechanism gives an efficient way to retrieve a data among a large set of nodes but cannot be used to manage more than one copy of each data.

The Data Manager is the component of DTM which manages the data. It can store persistent data, remove them or move them to another Data Manager on the DIET hierarchy. This component is only launched on the SeDs and uses all the available resources on the machine. Since DTM cannot manage more than one copy of a data, a persistent data cannot be copied from a SeD to another one. When a SeD asks for a data already stored on the hierarchy, the data is moved from its source (i.e., copied, then deleted). However, DTM allows the user to define a data as *sticky*: the data cannot be moved after they are stored on a node. Using this data persistence mode ensure the data will always be available where they have been stored. If another SeD needs a data, DTM copies it on this new storage and does not remove it from the first node. But the location manager "forgets" the previous data location. So, because the Location Manager cannot manage more than one data replica, such a replica will never be deleted or updated since the only SeD which knows it is the SeD which stored it. So, this data management mechanism can be very hazardous to use because the minimum data coherence cannot be ensured. Moreover, since only one data replica is known by the system, it is the only source available for a new data transfer even if a "closer" Data Manager could send the data faster or avoiding useless bandwidth consumption.

DTM is an easy-to-use and efficient data management solution for simple services. But its limitations deprive the users of the possible benefits of an advanced Grid data management. To avoid multiple transmissions of the same data from a client to a server, the DTM allows to leave data inside the platform after computation while data identifiers will be used further by the client to reference its data.

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 6.1.

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

Table 6.1: Persistence Modes.

In order to avoid interlacing between data messages and computation messages, the proposed architecture separates data management from computation management. The Data Tree Manager is build around three entities, the logical data manager, the physical data manager, and the data mover (see Figures 6.4 and 6.5).

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.

The Physical Data Manager is composed of a set of DataManager objects. The DataManager is located onto each SeD with which it communicates locally. It owns a list of persistent data. It stores data and has is charge of providing data to the server when needed. It provides features for data movement and it informs its LocManager parent of updating operations performed on its data (add, move, delete). Moreover, if a data is duplicated from a server to another one, the copy is set as non persistent and destroyed after it uses.

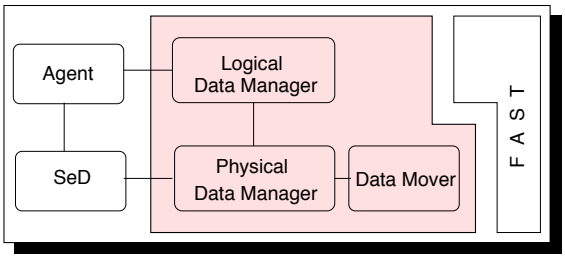


Figure 6.4: DTM: Data Tree Manager.

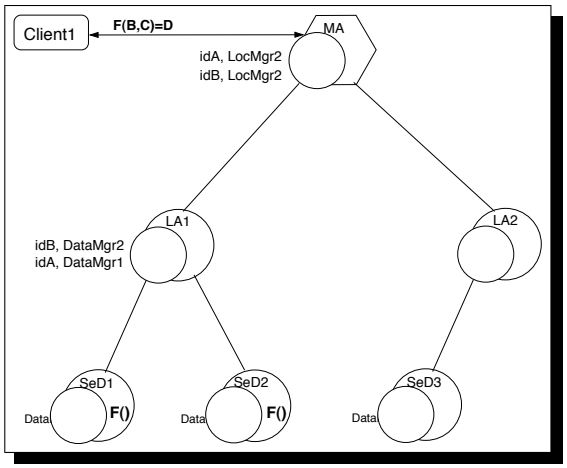


Figure 6.5: DataManager and LocManager objects.

This structure is built in a hierarchical way as shown in Figure 6.5. It is mapped on the DIET architecture. There are several advantages to define such a hierarchy. First, communications between agents (MA or LA) and data location objects (LocManager) are local like those between computational servers (SeD) and data storage objects (DataManager). This ensures a lower cost for the communication 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 on the 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 also has to initiate updates of DataManager and LocManager when a data transfer has finished. This data manager is no longer maintained due to lack of human resources.

6.3 JUXMEM

Joint work with:
- ⋆ Gabriel Antoniu : INRIA. IRISA Research Unit. Rennes. France.
- ⋆ Frédéric Desprez : INRIA. LIP Laboratory. Lyon. France.
- ⋆ Mathieu Jan : INRIA. IRISA Research Unit. Rennes. France.

An alternative data manager has been designed during the ACI project called GDS [15]. The aim of this data manager called JUXMEM is to provide a service of transparent data access. The service also transparently applies adequate replication strategies and consistency protocols to ensure data persistence and consistency in spite of node failures.

JUXMEM (Juxtaposed MEMory) [17] is a peer-to-peer data sharing service for the Grid. This service uses JXTA [163], a network programming and computing platform from Sun Microsystem, and it needs its own architecture deployment. JUXMEM can be used for data management in DIET: on the client side, the data needed for the DIET service execution are inserted into the

JuxMem service. Then, the data unique identifier is stored on the problem profile sent to the SeD, which directly asks JuxMem to obtain the data. This data management approach is interesting since JuxMem transparently ensures the data coherency, the persistence of the data and fault tolerance. However, using JuxMem, the user cannot decide where the data will be stored and it is difficult to evaluate the time needed to transfer a data to a specific node. So, JuxMem is a very interesting approach for the data management in Diet but cannot be used to implement some specific scheduling algorithms which need to explicitly distribute and replicate the data. Let us also mention that the interaction between JuxMem and Diet is still an experimental work.

To illustrate how a GridRPC system can benefit from transparent access to data, we have implemented the proposed approach inside the Diet GridRPC middleware, using the JuxMem data-sharing service. Note however that the concept of Grid data-sharing service can also be used in connection with other GridRPC middleware.

How Diet uses JuxMem to manage data? In our work, Diet internally uses JuxMem whenever a data is marked as persistent. However, we distinguish two cases for persistent data.

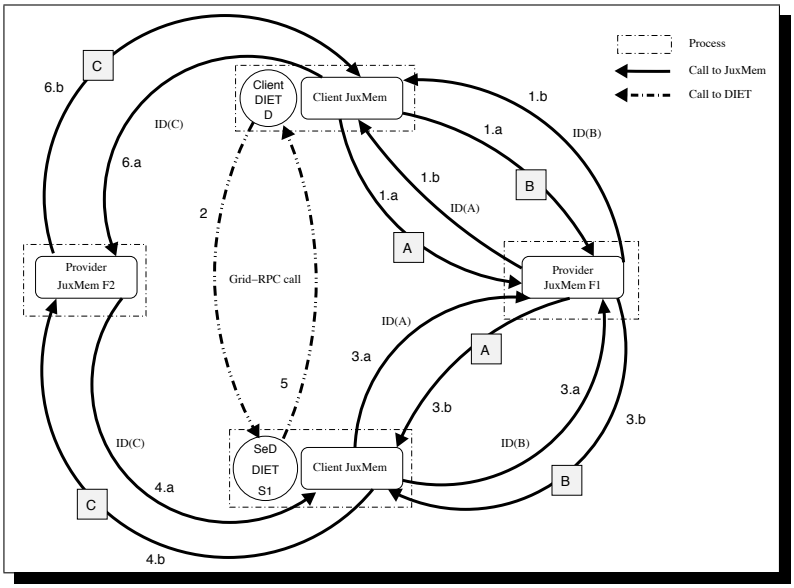


Figure 6.6: Multiplication of two matrices by a Diet client configured to use JuxMem for persistent data management.

Table 6.2 summarizes the interaction between Diet and JuxMem in each case, depending on the data access mode (e.g. `in`, `inout`, `out`) on both client/server sides. In the previous example, matrices A and B are `in` data, and matrix C is an `out` data. Note that for `inout` and `out` data, calls to JuxMem are executed after the computation on the client side only if the persistent mode is `PERSISTENT_RETURN`.

Modifications performed inside the Diet GridRPC middleware to use JuxMem for the management of persistent data are small. In our setting, Diet clients or SeDs use JuxMem's API to store/retrieve data, thereby acting as JuxMem clients. Also, note that our solution supports GridRPC interoperability, Diet simply uses JuxMem's API, with no extra code for data management.

	Client side		SeD side	
Computation	Before	After	Before	After
`in`	`attach; msync; detach;`		`mmap; acquire_read;`	`release; unmap;`
`inout`	`attach; msync;`	`acquire_read; release;`	`mmap; acquire;`	
`out`		`mmap; acquire_read; release;`		`attach; msync; unmap;`

Table 6.2: Use of JuxMem inside Diet for `in`, `inout` and `out` persistent data on client/server side, before and after a computation. The `juxmem` prefix has been omitted.

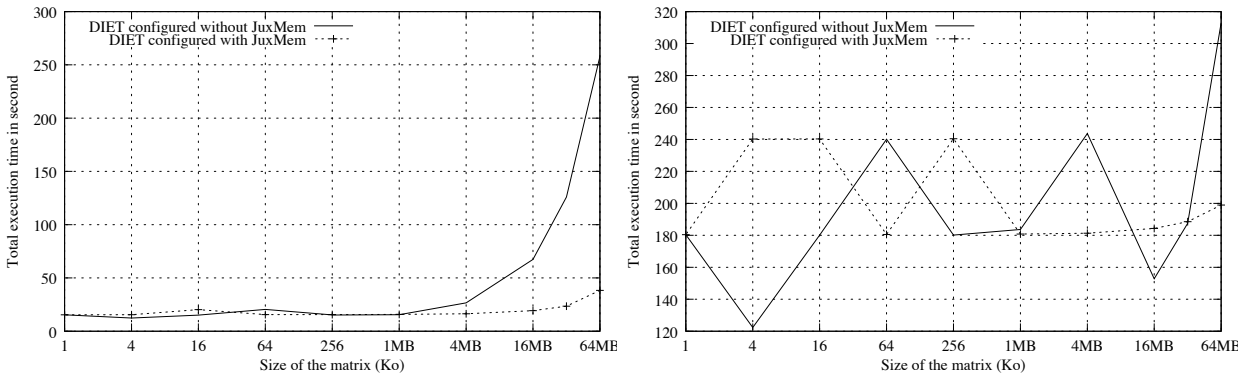


Figure 6.7: Total execution time (t) of a test with one persistent matrix of variable size when Diet is configured with and without JuxMem. The execution time of the service is s ($s = 5$ seconds on the left and $s = 60$ seconds on the right).

We now present the experimental evaluation of our JuxMem-based data management solution inside Diet.

We performed tests using 4 clusters (Rennes, Orsay, Toulouse and Lyon) of the French Grid'5000 testbed [41], using a total number of 3 sites simultaneously for a total number of 129 nodes.

Benefits of persistence: results based on a synthetic service. The goal of this test is to demonstrate and measure the benefits of the management of persistent data by JuxMem, in terms of impact on the overall execution time of a client's series of service invocations. For this test, one client located in one cluster of the Grid (here: Rennes) performs a series of 32 asynchronous GridRPC calls to a simple synthetic service.

Figure 6.7 shows the total execution time t of the client code when Diet is configured with and without JuxMem for the management of persistent data ($s = 5$ seconds on the left and $s = 60$ seconds on the right). For $m = 64$ MB, t is equal to 257 seconds when Diet is configured without JuxMem (with $s = 5$ seconds). This time is lowered to 38 seconds when Diet is configured with JuxMem (still with $s = 5$ seconds). This is a speedup of 6.8. The speedup start to become

noticeable for values of m higher than 1 MB. Note that for very small values of m, the overhead of using JuxMem, instead of directly sending data to SeD, is low (the value of t is essentially the same in both cases).

In addition to offer new functionalities for the data management into Diet as such as replication strategies and consistency protocols, JuxMem proofs that Diet can integrate easily alternative data manager. The IRISA does not maintain this data manager.

6.4 Dagda: Data Arrangement for Grid and Distributed Applications

Joint work with:
- ⋆ Gaël Le Mahec : CNRS/IN2P3 Clermont-Ferrand. France.
- ⋆ Frédéric Desprez : INRIA. LIP Laboratory. Lyon. France.

Dagda was first introduced in Diet version 2.3. It was designed to answer the different problems presented in Section 6.1. The first target application of Dagda was the Blast bioinformatics application which uses large databanks in read-only access and a very large number of short time requests. In this section, we introduce the Dagda data management capabilities and the different problems it can deal with.

6.4.1 Dagda: A new data manager for the Diet middleware

Dagda introduces some data management features that were lacking in the previous Diet data manager. It allows the Diet application developers to perform explicit data replications as well as implicit ones. It also introduces some optimizations for the storage resources and the transfers performances. It introduces automatic data management policy selection to choose which data can be erased to free space to store a new data. When choosing Dagda as the Diet data manager, the users can directly take benefit of the transfers optimizations and data replications without any change in their applications. But Dagda also introduces direct data management extending the standard Diet API.

Moreover, by using the Dagda API, the Diet users can manage directly their data:

- Store a data on the Grid, leaving Dagda choose where the data should be recorded.
- Retrieve a data, using its unique ID, leaving Dagda choose the best source node.
- Replicate a data by choosing where to copy it.
- Make a "snapshot" of the Grid data state, allowing to stop the middleware and restart it later with the same data distribution.
- Define a data ID "alias" to easily share a recorded data.

Dagda also introduces an advanced configuration of the nodes for the data management. The site administrator can limit the memory and disk space used by Dagda to store the data. He can choose where the data files will be stored and if the storage directory is accessible to the children of the node. Then, when a file is stored on such a configured agent, all of its children can access it transparently. Dagda is a part of the Diet middleware written in C++ language and relying

on Corba. It is optionally compiled with DIET as its data manager and it does not need special requirements. DAGDA will be the default data manager in the next release of DIET v3.0. Strong relationships between DAGDA and the workflow engine exist as we will see Section 7.3.

6.4.2 The DAGDA architecture

Figure 6.8 presents the DAGDA components architecture.

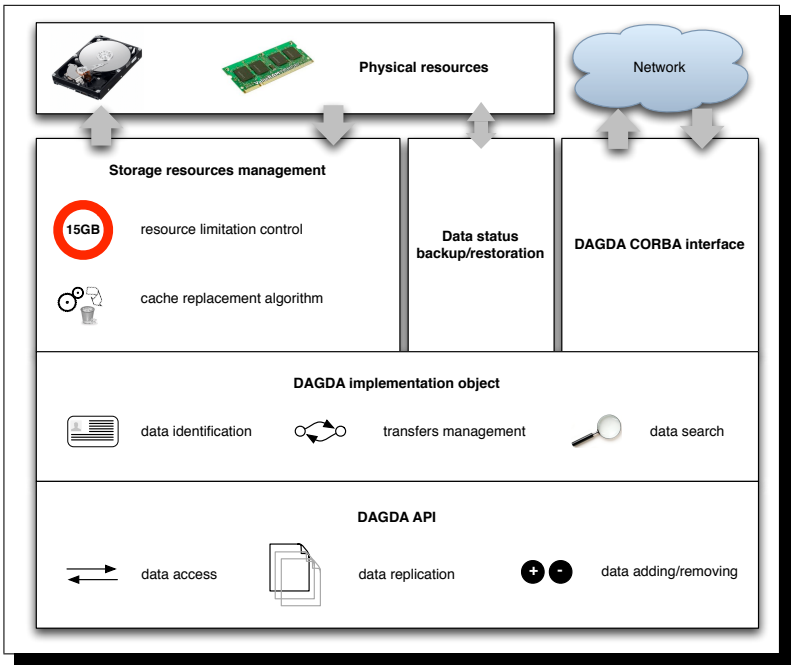


Figure 6.8: The DAGDA components internal architecture.

Storage resource management

Using DAGDA as DIET data manager the sites administrators can choose to fix storage resource limitations for the DIET services. These limitations can be fixed as well for the disk space as for the memory space that DAGDA can use to store the DIET data. Because of these explicit or implicit limitations (the total available disk and memory available space), the site administrator can choose one data replacement algorithm to use to select which data can be removed to free space to store a new data. Currently, DAGDA proposes three different strategies:

Least Frequently Used (LFU): The data which is the least frequently used, among the locally stored data having a sufficient size, is selected to free some space for a new data.

Last Recently Used (LRU): The data which was the last recently used, among the locally stored data having a sufficient size, is selected to free some space for a new data.

First In First Out (FIFO): The data of sufficient size which was recorded the longest time ago is selected to free some space for a new data.

Of course, it is possible to choose to never remove a data to free space for new data.

The DAGDA object

DAGDA uses a decentralized naming service. It uses a Universal Unique ID (UUID) library based on the RFC 4122 to generate identifiers. Using this identifier, DAGDA can retrieve a data contacting the others DAGDA components of the platform. Then, the data transfers can be directly initiated from each DAGDA component in the platform to all the others. DAGDA transfers the data in several parts. The maximum size of a the data chunk transmitted through the network at a time can be fixed in the DAGDA configuration. By limiting the size of the data chunks, the user can also limit the system memory that DAGDA will use to perform the transfers.

The DAGDA Corba interface

In its current version, DAGDA uses Corba to perform its communications and transfers through the network. The data search and remove commands are performed as simple Corba calls. The data transfers to add or update a specified data are performs through several Corba calls transmitting the data as parameter. For the data transmissions, DAGDA uses the *pull model*: the destination DAGDA components ask for the beginning of data transfer, in contrary of the *push model* where the data are directly transmitted from the source to the destination.

The API proposes functions to put or get data from/to the platform in a synchronous or asynchronous manners. The asynchronous transfers can be used to perform several data transfers in parallel but also to proceed to data prefetching. In the first case, the user must wait for the end of the transfers, calling a waiting function for each of them. In the second case, the transfers are done independently, and there is no need to wait for their ends. Indeed, by making some prefetching the user can anticipate the use of a data on a node, but waiting for the transfers to be completed partially decreases the prefetching benefits by also waiting for the useless transfers (in the end, the node might not be selected for the service execution).

The user can implement many replication strategy algorithms as long as he has sufficient information about the platform. In the next version of DAGDA, we will introduce more complex rules to allow the usage of conditional replication rules based on several characteristics of the nodes.

6.4.3 Interactions between DIET and DAGDA

Figure 6.9 presents interactions between DAGDA during a DIET service call.

1. The client performs a DIET call.
2. DIET selects one or more SeDs to execute the service.
3. The client submits its request to the selected SeD, sending only the data descriptions.
4. The SeD downloads the new data from the client and the persistent ones from the nodes on which they are stored.
5. The SeD executes the service using the input data.
6. The SeD performs updates on the inout and out data and sends their descriptions to the client. The client then downloads the volatile and persistent return data.

Each DIET component (clients, agents and SeDs) has its own DAGDA component and can interact with the complete DAGDA deployment. Figure 6.10 present the internal interactions between

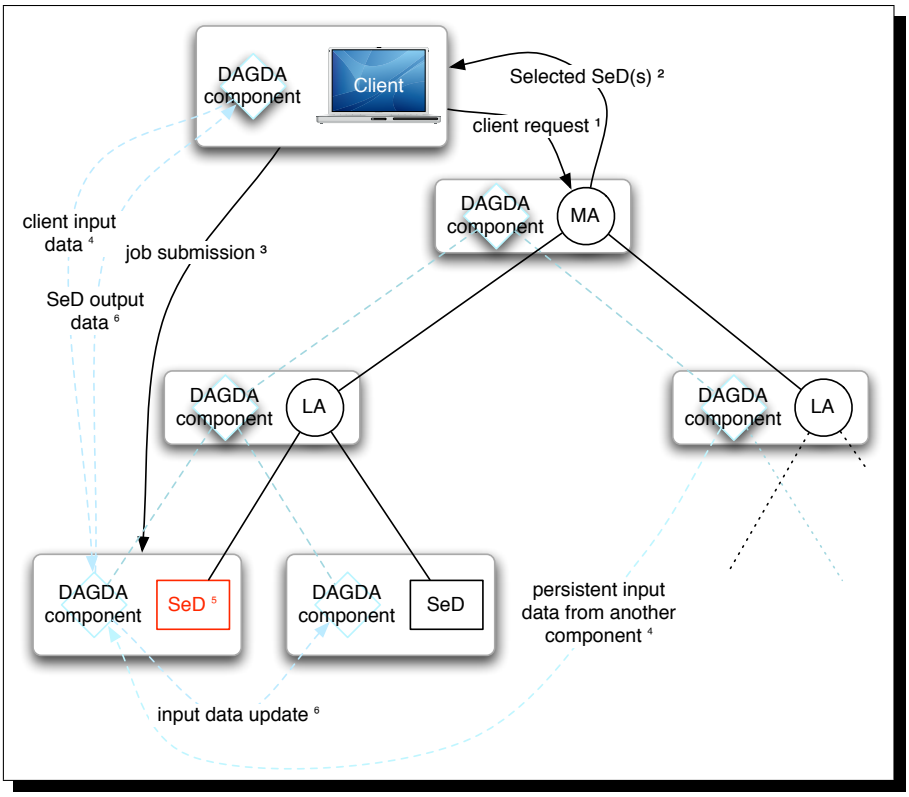


Figure 6.9: The DAGDA interactions with DIET.

the client or server applications and DAGDA. Figure 6.11 presents the interactions between a DIET agent and its DAGDA component.

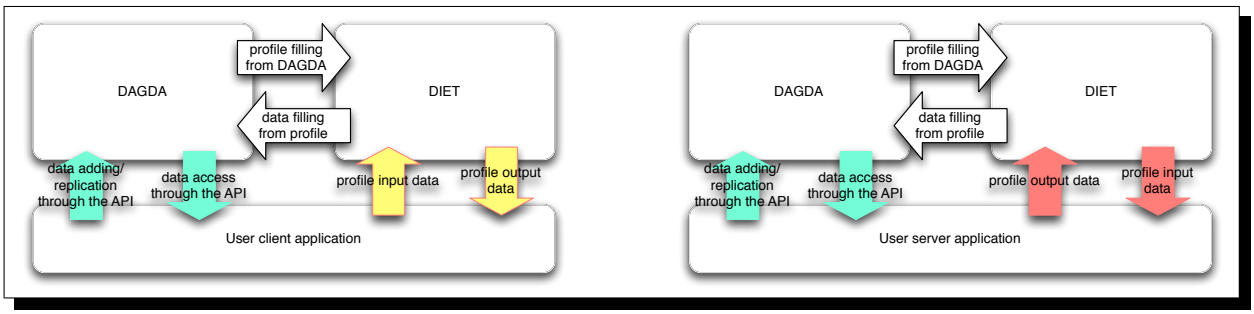


Figure 6.10: Interactions between DIET applications and DAGDA.

6.5 Conclusion

In this chapter we have shown how DIET deals with the data management. We have presented the GridRPC data management model and the corresponding API, which is now an Open Grid Forum standard. It is a simple, powerful, flexible, and effective means to manage data in a Grid

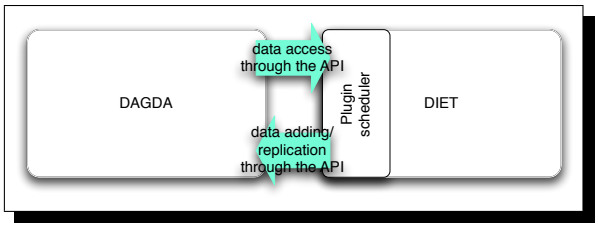


Figure 6.11: Interactions between DIET agents and DAGDA.

environment. We introduced three data managers for the DIET middleware which provide mechanisms for data persistency and replication. The use of these data managers can largely improve the global performances of the platform as described in [75]. Data management is fundamental for a Grid middleware. The knowledge established with DTM and JUXMEM has been very fruitful to design DAGDA. In the future, DAGDA will propose a loadable module support for data replacement algorithms to allow the users to easily develop by themselves their own algorithm to be used by DIET.

Chapter 7

Workflow management

Contents

A large number of scientific applications are represented by graphs of tasks which are connected based on their control and data dependencies. The workflow paradigm on Grids is well adapted for representing such applications, and the development of several workflow engines [14, 136, 154, 160] illustrate significant and growing interest in workflow management within the Grid community. The success of this paradigm in complex scientific applications can be explained by the ability to describe such applications with a high level of abstraction and in a way that makes it easy to understand, change, and execute them.

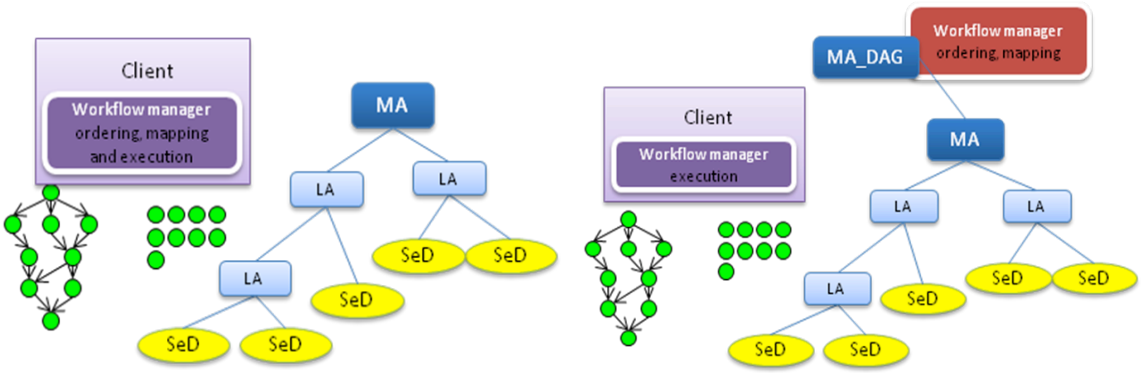
Several techniques have been established in the Grid community for defining workflows. The most commonly used model is the graph and especially the Directed Acyclic Graph (DAG). Since there is no standard language to describe scientific workflows, the description language is environment dependent and 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 high level API, the ability to use different advanced scheduling algorithms, and it allows the management of multi-workflows sent concurrently to the DIET platform.

7.1 Workflow architecture into DIET

Joint work with:
⋆ Abdelkader Amar : ENS Lyon. LIP Laboratory. Lyon. France.
⋆ Frédéric Desprez : INRIA. LIP Laboratory. Lyon. France.
⋆ Benjamin Isnard : INRIA. LIP Laboratory. Lyon. France.

DIET users, following the GridRPC paradigm, usually submit individual tasks. Workflows can of course be decomposed into individual tasks, but the knowledge of their overall structure their graphs helps the scheduler to make intelligent mapping decisions. Thus, we extended the agent

hierarchy by adding a new special agent to handle workflow submissions. This special agent, called MA_{DAG}, manages the different workflow submissions. An overview of the new DIET architecture is shown in Figure 7.1.



(a) Architecture 1: Complete schedule from the Client.

(b) Architecture 2: Schedule is handled by the MA_{DAG}

Figure 7.1: Two different architectures of DIET workflow engine.

The workflow support in DIET includes three modes. These three modes are represented by two architectures (Figure 7.1(a) and 7.1(b)) which are presented below, and can be used in the same platform without conflict.

- The first one provides a workflow manager at the client side. The client reads and processes the workflow description; a DAG structure is created. The client sends a set of problem descriptions to the MA to check if all services are available. If all services are available, the client starts the workflow execution. In this mode, since all scheduling operations are done in the client side, we can use a customized scheduler. The client programmer can write his personal scheduler.

- The second one uses a special agent called MA_{DAG}, which is responsible to communicate with the MA of the platform and provide an ordering and mapping for workflow execution.

- The third mode is similar to the second one, but in this mode, the MA_{DAG} only provides an ordering for the workflow execution.

The two architectures presented in Figure 7.1 can be used within the same DIET platform. The use of the MA_{DAG} is based on the user's 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 easily test 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-workflows can be handled more efficiently using core heuristics. To avoid overloading due to multiple workflow submissions from different clients, the MA_{DAG} is not responsible for workflow execution but it only manages the scheduling phase.

7.2 The workflow model

Joint work with:
⋆ Frédéric Desprez : INRIA. LIP Laboratory. Lyon. France.
⋆ Benjamin Isnard : INRIA. LIP Laboratory. Lyon. France.
⋆ Johan Montagnat : CNRS. I3S Laboratory. Sophia-Antipolis. France.

Because of large amounts of computations and data involved in some workflow applications, the number of tasks in a DAG can grow very fast. The need for a more abstract way of representing a workflow that separates the data instances from the data flow has led to the definition of a "functional workflow language" called the *Gwendia language* [127] designed by the GWENDIA ANR project, [1] in which the DIET team is involved.

A complex application can be defined using this language that provides data operators and control structures (if/then/else, loops, ...). To execute the application, we need to provide both the workflow description and a file describing the input data set. The DIET workflow engine will instantiate the workflow as one or several DAGs, sent to the MA_{DAG} agent to be executed in the DIET platform.

The GWENDIA workflow language describes a graph of "workflow nodes" where the vertices can belong to one of the following categories:

- computing "activities": a node describes a service that can be executed on a computing platform (e.g., a Grid or Cloud). The properties of this node include the data input and output ports with the type of data they receive or send.
- workflow control structures: a node controls the way other workflow nodes are used by the workflow engine. For example a conditional structure (if/then/else) can be used to trigger different activities depending on the value of an expression. These nodes also contain data input/output ports like activities.
- data "sources" and "sinks": these nodes produce data items (respectively receive them). A data source can be implemented as a file or as a database query. A data sink performance can be implemented as a terminal output or as a database insertion. These nodes usually define one output port (respectively one input port).

The edges of the graph are oriented, (one to one) and represent data flows between workflow nodes. More precisely, they interconnects ports of the workflow nodes together (one edge or "link" connects an output port to an input port).

The GWENDIA workflow language adopts a "data-driven" approach which means that the data model determines the execution model of the workflow [127]. This approach is different from control-driven workflow languages that use control structures that are independent from the data model to determine the execution schedule of the workflow.

This approach consists in using the characteristics of the data flows between workflow nodes to determine how the workflow is "instantiated", i.e., how the workflow engine generates a graph which has tasks as vertices and data dependencies (i.e., data "A" is produced by one task and used by another one) as edges. This graph can be generated dynamically and executed simultaneously, or the execution can be handled by a different workflow engine that takes this graph of tasks as input: a DAG. This engine manages task scheduling and execution.

[1] `http://gwendia.polytech.unice.fr`, contract ANR-06-MDCA-009.

Figure 7.2 shows the proposed architecture for the workflow engine. It shows the main components and their main dependencies

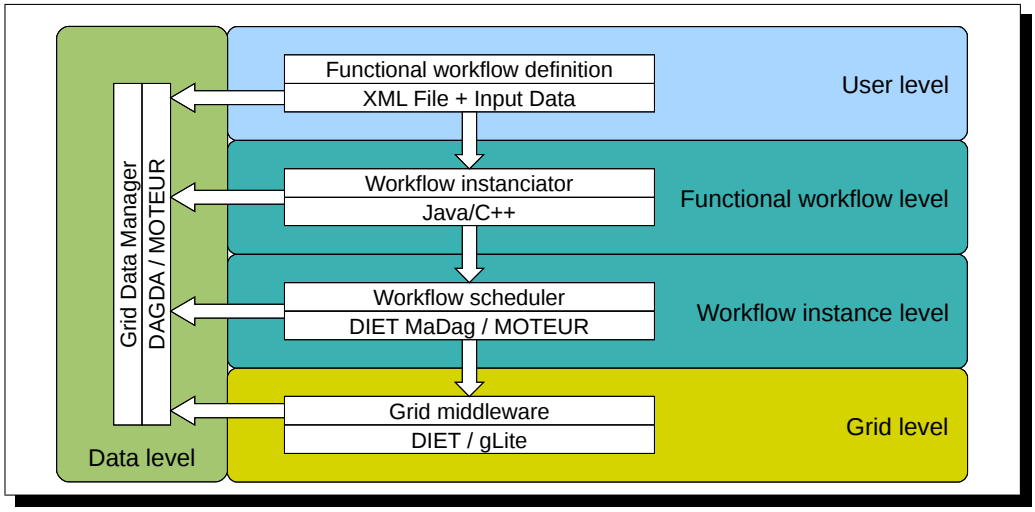


Figure 7.2: Architecture for GWENDIA language enactment.

7.3 Data management in workflow context

Joint work with:
- ⋆ Frédéric Desprez : INRIA. LIP Laboratory. Lyon. France.
- ⋆ Gaël Le Mahec : CNRS/IN2P3 Clermont-Ferrand. France.
- ⋆ Benjamin Isnard : INRIA. LIP Laboratory. Lyon. France.

The DAGDA system, described Section 6.4, provides a concept called "container" used to group several data together as a single data. A container is a logical data that can contain an arbitrary number of data elements. Its size is dynamically adjusted and there are no constraints on the types of data that can be inserted as elements. This can be viewed as a dynamic array of pointers to data elements. Containers, as any other data items managed by DAGDA, can be transferred between any DAGDA agents. But the transfer of the container does not necessarily mean the transfer of all its elements. This is only when an element is used that the transfer happens.

Using this concept, the workflow enactment engine can implement arrays of arbitrary depth and manipulate elements within the array, for example use references of elements of the array to transfer them individually. A container produced by a given task executed on a node does not necessarily need to be completely transferred to other processing nodes; in the case of data parallelism, only one element of the container is transferred to each processing node. Therefore it is essential that the workflow engine can provide a reference to that element only as the input of another task. The container is virtually splitted in different elements by the workflow engine.

Similarly, when a task requires consolidation of results from many other tasks, usage of containers avoids data transfers through the client that would be required to build the input data. With containers the input data is virtually containing all the outputs to process, but the data transfers occurs effectively only when required and between processing nodes only.

7.4 Conclusion

With the DIET workflow extension, we have provided a solution to easily submit a set of programs with dependencies, called with in a single request. We have developed this approach to serve the needs of applications, among them we can cite the cosmology application described in [39], Bio-informatics applications as PipeAlign (product a high quality protein alignment), Gee (annotate human genes according to not only their expression in neurological or muscular tissues, but also the expression of their homologue in other species), and, Maxdo (used to detect protein-protein and protein-DNA interactions.). Figure 7.3 shows the associated workflow of applications cited above. All of three have been experimented with DIET in [33].

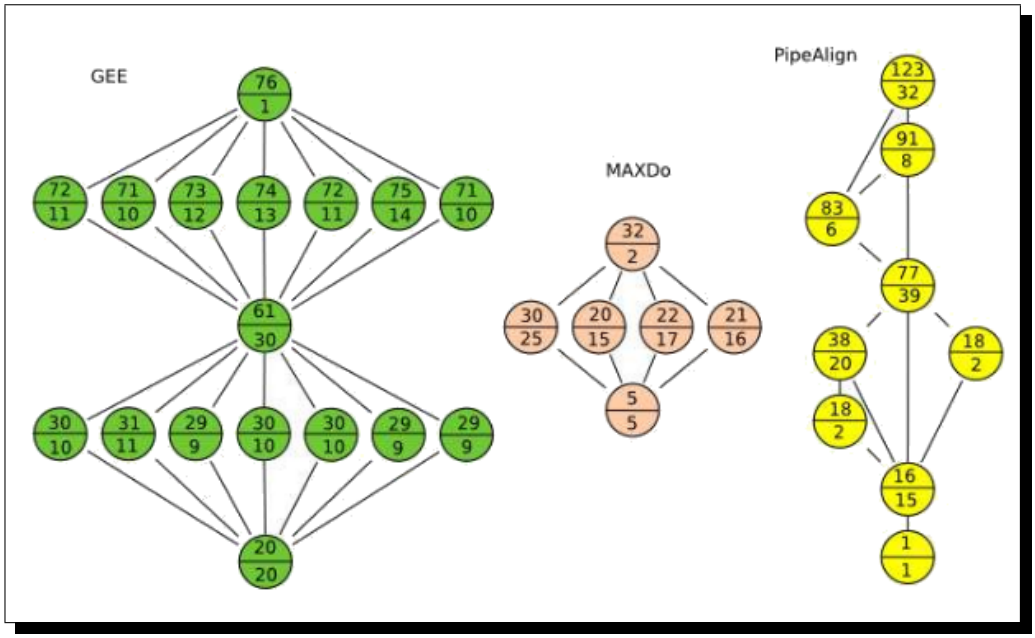


Figure 7.3: Shape of the workflow for three bio-informatics application.

The MA_{DAG} engine offers a DAG generator engine built on top of DIET. MA_{DAG} benefits from all the workflow scheduling mechanisms included in it. To provide a better reactiveness it greedily generates partial DAGs, these sub-DAGs are treated as soon as possible. In complete nodes are produced on the unforeseeable constructs until the execution engine reaches these points and they can be resolved. More sub-DAGs are then produced until completion of the execution.

More recently, we have used the workflow engine to provide Map/Reduce [70] mechanisms with DIET.

Chapter 8

Scheduling

Contents

The main interest and functionality of a Grid middleware is to hide the resource access. It is not so difficult to provide this access, the real challenge resides in the quality of the chosen resources. As introduced in Chapter 2, with the GridRPC paradigm, 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. The internal quality of a Grid or Cloud middleware is related to the quality of the scheduler. Moreover, to make effective use of today's scalable resource platforms, it is important to ensure scalability in the middleware layers, the scheduling is one of them.

Different works have been done on this problem. Among them, APST [56] allows internals modifications on the internals of the scheduling phase, mainly to be able to choose the scheduling heuristic. Several heuristics can be used like Max-min, Min-min, or X-sufferage (if necessary reader can refer to [57] to more details about these heuristics). Scheduling heuristics for different application classes have been designed within the AppLeS [58] and GrADS [27] projects. GrADS is built upon three components [66]. A Program Preparation System handles application development, composition, and compilation, a Program Execution System provides on-line resource discovery, binding, application performance monitoring, and rescheduling. Finally, a binder performs a resource-specific compilation of the intermediate representation code before it is launched on the available resources. One interesting feature of GrADS is that it provides application-specific performance models. Depending on the target application, these models can be extended, based

on analytical evaluations by experts and empirical models obtained by real world experiments. For some applications, simulations may be used to project their behavior on particular architectures. These models are then fed into the schedulers to find the most appropriate resources used to run the application.

Some work has been done to be able to cope with dynamic platform performance at the execution time [146, 166]. These approaches allows an algorithm to automatically adapt itself at run-time depending of the performance of the target architecture, even if it is heterogeneous.

Within the Condor project [162], the ClassAds language was developed [143]. The syntax of this language is able to express resource query requests with attributes through a rich set of language constructs: lists of constants, arbitrary collections of constants, and variables combined with arithmetic and logic operators. The result is a multi-criteria decision problem. The approach presented in our manuscript allows a scheduling framework to offer useful information to a metascheduler like Condor.

8.1 DIET Distributed Scheduling

As we mentioned in the introduction of this chapter, 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 makespan 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.

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 requested service 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 sub-trees during the deployment process. When a SeD or agent starts, 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 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 a 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

Network Weather Service (NWS) [173] or can provide application-specific performance predictions using the performance evaluation module FAST [140, 141] (see Section 8.5.1).

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 that a complete 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).

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 to provide 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.

8.2 Scheduling Extensions

Joint work with:
⋆ Holly Dail : INRIA. LIP Laboratory. Lyon. France.
⋆ Frédéric Desprez : INRIA. LIP Laboratory. Lyon. France.
⋆ Alan Su : INRIA. LIP Laboratory. Lyon. France.

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 (MA) levels [65].

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 perform. Such control at the server-level is also necessary to support some distributed scheduling approaches of interest. The following paragraph shows through an example which kind 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 a **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. Figure 8.1 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

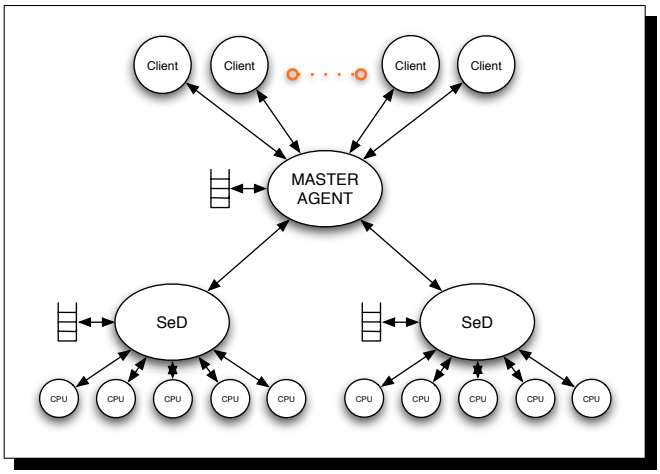


Figure 8.1: DIET extensions for request flow control.

executing we individually store and track 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 can not 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 can not 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 problem but not least, relates to fault-tolerance. If a server crash occurs, we also lose the queue information. Thus, a replication mechanism should be implemented.

At the MA 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 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 GlobalTaskManager must schedule some jobs in order to have information about server loads and queue lengths. Information may given from NWS sensor [173], as discussed in Section 8.5.

8.3 Plugin Schedulers

Joint work with:
⋆ Frédéric Desprez : INRIA. LIP Laboratory. Lyon. France.
⋆ Alan Su : INRIA. LIP Laboratory. Lyon. France.

At the beginning basic scheduling in DIET was based on the FIFO principle – a task submitted by a client was scheduled on the SeD whose response to the service query arrived the first to the MA. This approach did not directly take into consideration the dynamic state of the SeDs. A second version allowed a mono-criteria scheduling based on application-specific performance predictions. Later a round-robin scheduling scheme was implemented in DIET, resulting in good performance for task distribution over a homogeneous platform; however, this approach was found to be inefficient when the workload or the platform was sufficiently heterogeneous.

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

With our recent enhancements of the DIET toolkit, applications are now able to exert a degree of control over the scheduling subsystem via **plug-in schedulers**. As the applications that are to be deployed on the Grid vary greatly in terms of performance demands, the DIET plug-in scheduler facility permits the application designer to express application needs and features in order that be taken into account when application tasks are scheduled. These features are invoked at runtime after a user has submitted a service request to the MA, which broadcasts the request to its agent hierarchy.

When an application service request arrives at a SeD, it creates a **performance estimation vector** – a collection of **performance estimation values** that are pertinent to the scheduling process for that application (cf Table 8.1). The values to be stored in this structure can either be values provided by CoRI (Collectors of Resource Information) (see Section 8.5.3), or custom values generated by the SeD itself. The design of the estimation vector subsystem is modular; future performance measurement systems can be integrated with the DIET platform in a fairly straightforward manner.

Information tag starts with EST_	multi-value	Explanation
TCOMP		the predicted time to solve a problem
TIMESINCELASTSOLVE		time since last solve has been made (sec)
FREECPU		amount of free CPU 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)
NETWORKBANDWIDTH		network bandwidth (Mb/sec)
NETWORKLATENCY		network latency (sec)
TOTALSIZEDISK		size of the partition (Mb)
FREESIZEDISK		amount of free place 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 8.1: Explanation of the estimation tags.

CoRI generates a basic set of performance estimation values, which are stored in the estimation vector and identified by system-defined tags; Table 8.1 lists the tags that may be generated by a standard CoRI installation. 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 an appropriate server may be identified for each client request. By default, when a user request arrives at a DIET SeD, an estimation vector is created via a default estimation function; typically, this function populates the vector with standard CoRI values. If the application developer includes a custom **performance estimation function** in the implementation of the SeD, the DIET framework will associate the estimation function with the registered service. Each time a user request is received by a SeD associated with such an estimation function, that function, instead of the default estimation procedure, is called to generate the performance estimation values.

In the performance estimation routine, the SeD developer should store in the provided estimation vector any performance data needed by the agents to evaluate and compare server responses. Such vectors are then the basis on which the suitability of different SeDs for a particular application service request is evaluated. Specifically, a local agent gathers responses generated by the SeDs that are its descendants, sorts those responses based on application-specific comparison metrics, and transmits the sorted list to its parent. The mechanics of this sorting process comprises an **aggregation method**, which is simply the logical process by which SeD responses are sorted. 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 multi-criteria scheduling policy based on a series of performance values that are to be optimized in succession.

The `diet_profile_desc_aggregator` and `diet_aggregator_set_type` functions fetch and configure the aggregator corresponding to a DIET service profile, respectively. 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.

To define the tags and the order in which they should be compared, four functions are introduced. These functions, of the form `diet_aggregator_priority_*`, serve to identify the estimation values to be optimized during the aggregation phase. The `_min` and `_max` forms indicate that a standard performance metric (e.g., time elapsed since last execution, from the `diet_estimate_lastexec` function) is to be either minimized or maximized, respectively. Similarly, the `_minuser` and `_maxuser` forms indicate the analogous operations on user-supplied estimation values. The order of calls to these functions indicate the order of **precedence** of the tags, with higher priority given to tags specified earlier in the process of defining the scheduler.

8.4 Workflow Scheduling

Joint work with[1]:
- ⋆ Raphaël Bolze : CNRS. LIP Laboratory. Lyon. France.
- ⋆ Frédéric Desprez : INRIA. LIP Laboratory. Lyon. France.
- ⋆ Benjamin Isnard : INRIA. LIP Laboratory. Lyon. France.

The MA_{DAG} agent may receive many requests to execute workflows from one or several clients, and the number of resources to execute all tasks in parallel may not be sufficient on the grid. In this case the choice of a particular workflow scheduler is critical to determine the order of execution of all tasks that are ready to be executed.

Schedulers provide different online scheduling heuristics that apply different prioritization algorithms to choose the order of execution between tasks of the same DAG (intra-DAG priority) and between tasks of different DAGs (inter-DAG priority). All heuristics are based on the well-known HEFT (Heterogeneous Earliest Finish Time) heuristic [164] that is extended to this case of online multi-workflow scheduling.

The available MA_{DAG} workflow schedulers are:

- A basic scheduler: this scheduler manages the precedence constraints between the tasks. The priority between tasks within a DAG is set according to the HEFT heuristic. When a task is ready to be executed (i.e., the preceding tasks are completed) the ready task with the higher HEFT rank is sent to the client for execution without specifying a resource. Then the client performs a standard DIET request that will use the scheduler configured by the SeD.
- A Multi-HEFT scheduler: this scheduler applies the HEFT heuristic to all workflows submitted by different clients to the MA_{DAG}. This means that the priorities assigned by the HEFT heuristic are used to order the tasks of all DAGs processed by the MA_{DAG} and following this order the tasks are mapped to the first available resource.
- A Multi-AgingHEFT scheduler: this scheduler is similar to Multi-HEFT but it applies a correction factor to the priorities calculated by the HEFT algorithm. This factor is based on the age of the DAG i.e., the time since it was submitted to the scheduler. Compared to Multi-HEFT this scheduler will increase the priority of the tasks of a workflow that has been submitted earlier than other DAGs.
- A FOFT (Fairness on Finish Time) scheduler: this scheduler uses another heuristic to apply a correction factor to the priorities calculated by the HEFT algorithm. This factor is based on

[1] My participation around this work was marginal. Nevertheless, to give a complete overview of DIET this work is briefly described in this manuscript.

the slowdown of the DAG that is calculated by comparing the earliest finish time of the tasks in the same environment without any other concurrent workflow and the actual estimated finish time.

The design of these algorithms and the evaluations of them are available in [30]

The workflow schedulers use information provided by the SeDs to be able to run the HEFT heuristic. So the SeD programmer must provide the required data in the estimation vector by implementing a plugin scheduler (see Section 8.3).

8.5 Performance evaluation

Scheduling tasks on computers comes down to mapping task requirements to system availability. We now describe these values more precisely. Requirements of routines group principally the time and the memory space necessary to their execution, as well as the amount of generated communication. These values 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 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.

8.5.1 FAST

Joint work with:
- ⋆ Frédéric Desprez : INRIA. LIP Laboratory. Lyon. France.
- ⋆ Martin Quinson : ENS Lyon. LIP Laboratory. Lyon. France.

The goal of FAST [140] is to constitute a simple and consistent Software Development Kit (SDK) for providing client applications with accurate information about task requirements and system performance information, regardless of how theses 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 8.2 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 in two submodules: the right one on the figure forecasts the system performance capabilities while the left one uses the routine's requirements models. Figure 8.2 shows that FAST uses principally two types of external tools (in grey): A system monitoring tool such as NWS [173], 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 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 (Network Weather Service) [173] is a project leaded by Pr. Wolski at the University of California, Santa-Barbara. It constitutes a distributed set of sensors and statistical forecasters

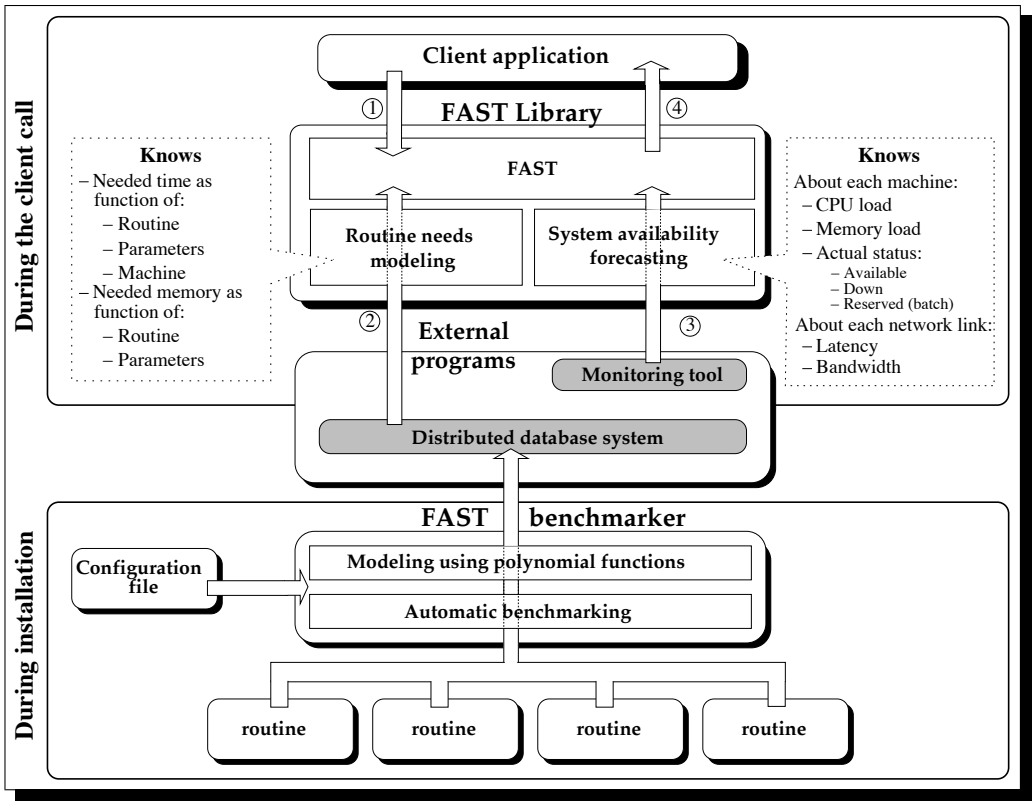


Figure 8.2: FAST's architecture.

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, NWS can not only report the current load, but also the time-slice a new process would get at startup. In order to benefit from a solid and well tested basis, the main monitoring system used is NWS. But for 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 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 latency and bandwidth of any TCP link. In addition to NWS, it can also report the number of CPUs on each host to ease the comparison. Monitoring new resources like 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 can not 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 in [78].

8.5.2 Performance evaluation for parallel program

Joint work with:
⋆ Frédéric Desprez : INRIA. LIP Laboratory. Lyon. France.
⋆ Frédéric Suter : ENS Lyon. LIP Laboratory. Lyon. France.

The computational servers we target can be sequential or parallel. Moreover, they are running libraries such as the BLAS, LAPACK and/or their parallel counterparts (PBLAS, ScaLAPACK). The latter libraries are using virtual grids of processors and block-sizes to distribute the matrices involved in the computations. The performance evaluation has thus to take into account the shape of the grid as well as the distributions of the input and output data. We propose a careful evaluation of several linear algebra routines combined with a run-time evaluation of the parameters of the target machines. This allows us to optimize the mapping of data-parallel tasks in a client-server environment. We have extended FAST. We propose a combination between dynamic data acquisition and code analysis which will be the core of the extension of FAST to handle parallel routines. We detail the extension on two examples of parallel routines.

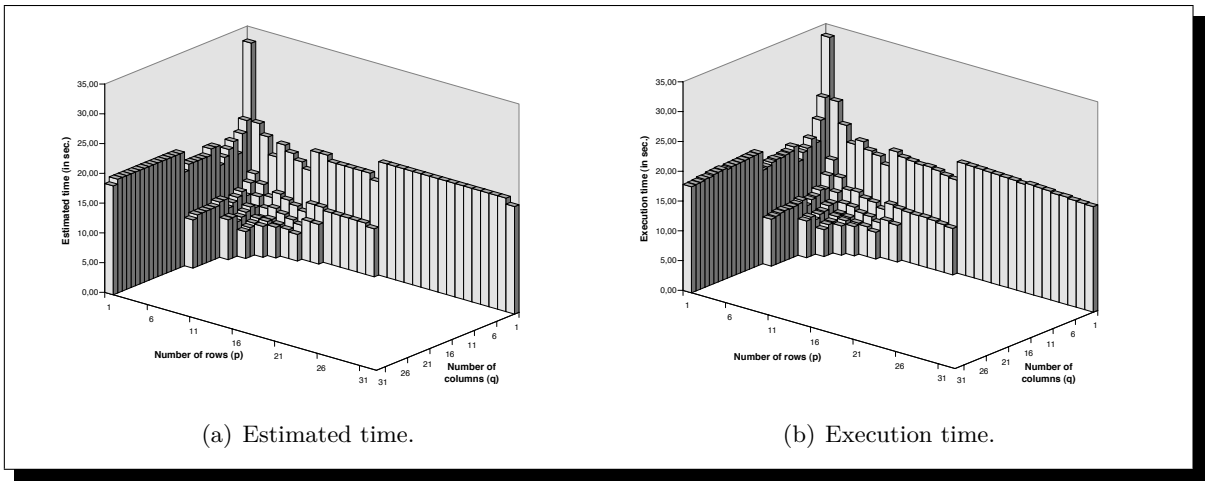
To obtain a good schedule for a parallel application it is mandatory to initially determine the computation time of each of its tasks and communication costs introduced by parallelism. The most common method to determine these times is to describe the application and model the parallel computer that executes the routine. We did this kind of modelization in [53].

Extension of FAST to Handle Parallel Routines

The first version of the extension only handles some routines of the parallel dense linear algebra library ScaLAPACK. For such routines the description step consists only in determining which sequential counterparts are called, their calling parameters (i.e., data sizes, multiplying factors, transposition, ...), the communication schemes and the amount of data exchanged. Once this analysis is completed, the computation part can easily be forecasted. Indeed, since FAST is able to estimate each sequential counterpart, FAST calls are sufficient enough to determine their execution times.

Furthermore, processors executing a ScaLAPACK code have to be homogeneous to achieve optimal performance, and the network between these processors also has to be homogeneous. It allows us two major simplifications. First, processors being homogeneous, the benchmarking phase of FAST can be executed on only one processor. Then concerning communications we only have to monitor a few representative links to obtain a good overview of the global behavior.

Figure 8.3 presents a comparison between the estimated time given by our model Figure 8.3(a) and the actual execution time Figure 8.3(b) for the `pdgemm` routine on all possible grids from 1 up to 32 processors. Matrices are of size 2048 and the block size is fixed to 64. The x-axis represents the number of rows of the processor grid, the y-axis the number of columns and the z-axis the execution time in seconds. We can see that the estimation given by our extension is very close to the experimental execution times. The maximal error is less than 15% while the average error is less than 4%. Furthermore, these figures confirm the impact of topology on performance. Indeed, compact grids achieve better performance than elongated ones because of the symmetric communication pattern of the routine. The different stages for row and column topologies can be explained by the log term introduced by the broadcast tree. These results shows that our evaluation can be efficiently used to choose a grid shape for a parallel routine call. Combined with a run-time evaluation of parameters like communication, machine load or memory availability, we can then



(a) Estimated time. (b) Execution time.

Figure 8.3: Comparison for the `pdgemm` routine on all possible grids from 1 up to 32 processors.

build an efficient scheduler for ASP environments.

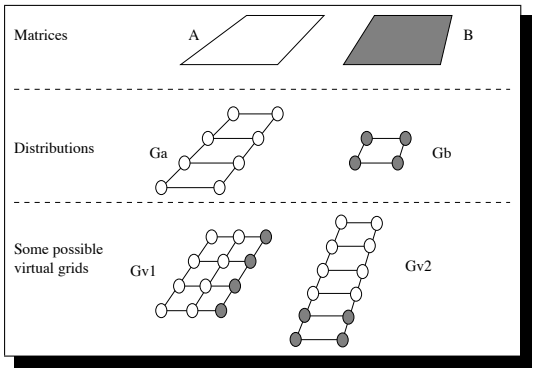


Figure 8.4: Initial distribution and processors grids used in this experiment.

The objective of our extension of FAST is to provide accurate information allowing at a scheduler, to determine which is the best solution among several scenarios. Let us assume we have two matrices A and B we aim to multiply. These matrices have the same sizes but distributed in a block-cyclic way on two disjointed processor grids (respectively G_a and G_b). In such a case it is mandatory to align matrices before performing the product. Several choices are then possible: redistribute B on G_a, redistribute A on G_b or define a new virtual grid with all available processors. Figure 8.4 summarizes the framework of this experiment. These grids are actually sets of nodes from a single parallel computer (or cluster). Processors are then homogeneous. Furthermore, inter– and intra–grids communication costs can be considered as similar.

Unfortunately the current version of FAST is not able to estimate the cost of a redistribution between two processor sets. This problem is indeed very hard in the general case [76]. So for this experiment we have determined the amounts of data transferred between each pair of processors and the communication scheme generated by the ScaLAPACK redistribution routine. Then, we use FAST to forecast the costs of each point to point communication. Figure 8.5 gives a comparison

between forecasted and measured times for each of the grids presented in Figure 8.4.

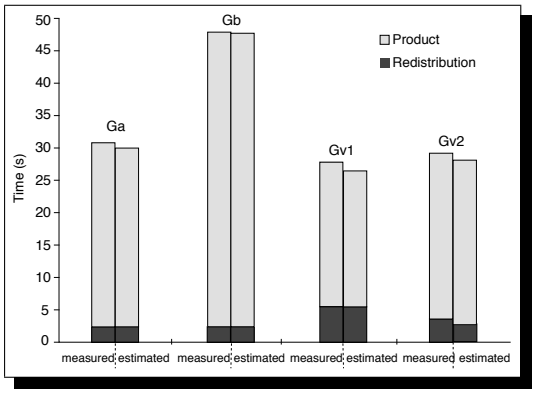


Figure 8.5: Validation of the extension in the case of a matrix alignment followed by a multiplication. Forecasted times are compared to measured ones distinguishing redistribution and computation (matrix size 2000×2000).

We can see that the parallel extension of FAST allows to accurately forecast what is the best solution, namely a 4×3 processor grid. If this solution is the most interesting with regards to the computation point of view, it is also the less efficient from the redistribution point of view. The use of FAST can then allow to perform a first selection depending on the processor speed/network bandwidth ratio. Furthermore, it is interesting to see that even if the choice to compute on G_a is a little more expensive, it induces less communications and releases 4 processors for other potential pending tasks. Finally a tool like the extended version of FAST can detect when a computation will need more memory than the available amount of a certain configuration and thus induce swap. Typically the 2×2 processor grid will no longer be considered as soon as we reach a problem size exceeding the total capacity of involved processors. For larger problem sizes the 4×2 grid may also be discarded.

This experiment shows that the extension of FAST to handle parallel routines will be very useful to a scheduler as it provides enough information to be able to choose according to several criteria: Minimum Completion Time (MCT), communication minimization, number of processors involved, ...

As we have seen FAST is a good way to provide the information to the scheduler. However when we develop a project as DIET it was too risky to base a critical aspect only on one tool. Moreover sometimes Grid platforms provide an integrated tool, e.g., Ganglia is deployed on a given platform. Then, to be compliant with many Grid Resource Information Services (GRIS), DIET must provide a tool to easily deal with any of GRIS. CoRI was designed with this goal.

8.5.3 CoRI

As we have seen in the previous section, the scheduler requires performance measurement tools to make effective scheduling decisions. Thus, DIET depends on reliable Grid resource Information Services. In this section, we introduce the exact requirements of DIET for a Grid information service, the architecture of the new tool CoRI (Collectors of Resource Information) and the different components inside of CoRI.

CoRI architecture

In this section, we describe the design of CoRI, this new platform performance subsystem that we have implemented to enable future versions of the DIET framework to more easily interface with third-party performance monitoring and prediction tools. Our goal is to facilitate the rapid incorporation of such facilities as they emerge and become widely available. This issue is especially pertinent, considering the fact that DIET is designed to run on heterogeneous platforms, on which many promising but immature tools may not be universally available. Such a scenario is common, considering that many such efforts are essentially research prototypes. To account for such cases, we have designed the performance evaluation subsystem in DIET to be able to function even in the face of varying levels of information in the system.

We designed CoRI to ensure that it (i) provides timely performance information to avoid impeding the scheduling process and (ii) presents a general-purpose interface capable of encapsulating a wide range of resource information services. Firstly, it must provide basic measurements that are available regardless of the state of the system. The service developer can rely on such information even if no other resource performance prediction service like FAST or NWS is installed. Secondly, the tool must manage the simultaneous use of different performance prediction systems within a single heterogeneous platform. To address these two fundamental challenges, we offer two solutions: the **CoRI-Easy** collector to universally provide basic performance information, and the **CoRI Manager** to mediate the interactions among different collectors. In general, we refer collectively to both these solutions as the **CoRI** tool, which stands for Collectors of Resource Information.

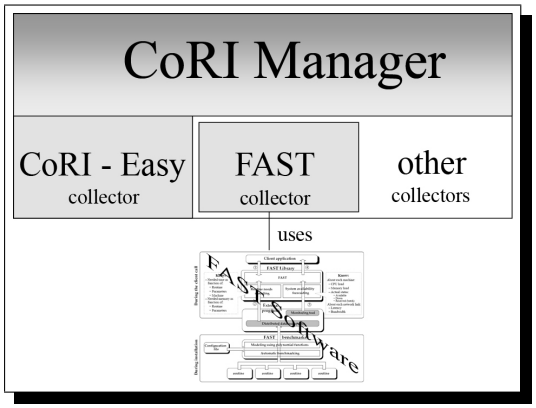


Figure 8.6: The CoRI architecture: The CoRI-Manager and its collectors, namely the CoRI-Easy and the FAST collector.

CoRI Manager

The CoRI Manager provides access to different **collectors**, which are software components that can provide performance information about the system. This modular design decouples the choice of measurement facilities and the utilization of such information in the scheduling process. Even if the manager should aggregate across performance data originating from different resource information services, the raw trace of data remains, and so its origin can be determined. For example, it could be important to distinguish the data coming from the CoRI-Easy collector and the FAST collector,

because the performance prediction approach that FAST uses is more highly tuned to the features of the targeted application. Furthermore, the modular design of the CoRI Manager also permits a great deal of extensibility, in that additional collectors based on systems, such as Ganglia [123] or NWS [173], can be rapidly implemented via relatively thin software interface layers. This capability enables DIET users to more easily evaluate prototype measurement systems even before they reach full production quality.

Estimation vector

In this section, we describe in greater detail the structure of the **estimation vector**, the data structure used within DIET to transmit resource performance data. We then enumerate the standard metrics of performance used in DIET and present the various CoRI Manager functions. The estimation vector is divided into two parts. The first part represents "native" performance measures that are available through CoRI (e.g., the number of CPUs, the memory usage, etc.) and the scheduling subsystem (e.g., the time elapsed since a server's last service invocation). The second part is reserved for developer-defined measurements that are meaningful solely in the context of the application being developed. The vector supports the storage of both singleton data values (which we call scalars) and data value series, because some performance prediction measurements are inherently a list of values (e.g., the load on each processor of a multi-processor node). An estimation vector is essentially a container for a complete performance snapshot of a server node in the DIET system, composed of multiple scalars and lists of performance data.

To differentiate among the various classes of information that may be stored in an estimation vector, a data tag is associated with each scalar value or list of related values. This tag enables DIET's performance evaluation subsystems to identify and extract performance data. There are two types of tags: system tags and user-defined tags. System tags correspond to application-independent data that are stored and managed by the CoRI Manager; Table 8.1 (page 95) enumerates the set of tags that are supported in the current DIET architecture. User-defined tags represent application-specific data that are generated by specialized performance estimation routines that are defined at compile-time for the SeD. At the moment of service registration, a SeD also declares a priority list of comparison operations based on these data that logically expresses the desired performance optimization semantics. The details of the mechanisms for declaring such an optimization routine fall outside the scope of this manuscript; for a fully detailed description of this API, please consult [161].

The basic public interface of the CoRI Manager that is available to DIET service developers consists of three functions. The first function allows the initialization of a given collector and adds the collector to the set of collectors that are under the control of the CoRI Manager. The second function provides access to measurements. The last function tests the availability of the CoRI-Easy collector.

CoRI-Easy collector

The CoRI-Easy collector is a resource collector that provides basic performance measurements of the SeD. Via the interface with the CoRI Manager, the service developer and the DIET developer are able to access CoRI-Easy metrics. We first introduce the basic design of CoRI-Easy, and then we discuss some specific problems. CoRI-Easy should be extensible like CoRI Manager, i.e., the measurement functions must be easily replaceable or extended by new functionality as needed.

Consequently, we use a functional approach: functions are categorized by the information they provide. Each logical type of performance information is identified by an information class, which enables users to simply test for the existence of a function providing a desired class of information. Thus, it is even possible to query the CoRI-Easy collector for information about the platform that may not yet have been realized. Our goal was not to create another sensor system or monitor service; CoRI-Easy is simply a set of routines that produce basic performance metrics. Note that the data measured by CoRI-Easy are available via the interface of the CoRI Manager.

CPU evaluation. CoRI-Easy provides CPU information about the node that it monitors: the number of CPUs, the CPU frequency and the CPU cache size. These static measurements do not provide a reliable indication of the actual load of CPUs, so we also measure the node's BogoMips (a normalized indicator of the CPU power), the **load average** and **CPU utilization** for indicating the CPU load.

Memory capacity The memory is the second important factor that influences performance. CoRI monitors the **total memory size** and the **available memory size**.

Disk Performance and capacity. CoRI-Easy also measures the **read and write performance** of any storage device available to the node, as well the **maximal capacity** and the **free capacity** of any such device.

Network performance. CoRI-Easy should monitor the performance of interconnection networks that are used to reach other nodes, especially those in the DIET hierarchy to which that node belongs; this functionality is planned for a future release.

In this part we have seen how CoRI can provide to DIET an access to different resource information services (i.e., FAST or CoRI-Easy). DIET can take advantage of the CoRI design that easily supports the growing number of resource information services. Thus, CoRI is a good way to add new collectors that are especially designed for the grid environment. So we can see the CoRI Manager as an open window that allows the use of a lot of different tools developed around the world, and the CoRI-Easy collector as the first version of a homemade resource monitor.

8.6 Conclusion

In this chapter, we described the design of the plug-in scheduler in DIET. To provide underlying resource performance data needed for plug-in schedulers, we designed tools that facilitates the access to the resources information. FAST provides an efficient forecasting tool requiring benchmarks of the applications. We have extended the capabilities of FAST to parallel programs. Finally we designed a tool to provide the management of different performance measures and different kinds of resource collector tools.

The plug-in facilities enable the scheduling information to be readily encoded with the application, eliminating the need for potentially expensive microbenchmarks. Moreover, the CoRI resource performance collection infrastructure enables middleware maintainers to closely assess and control the intrusiveness of the monitoring subsystem. By configuring CoRI to gather only the information that is needed for the applications that a specific DIET hierarchy is meant to support, excessive monitoring costs are effectively avoided. The performance estimations provided for scheduling are

suitable for dedicated resource platforms with batch scheduler facilities. In a shared resource environment, there may be differences between the estimations obtained at the SeD selection moment and the values existing when the client-SeD communication begins. However, even though in some cases the information could be marginally incorrect, an informed decision is preferable.

The plug-in scheduler is a powerful mechanism of DIET that makes DIET a middleware which can deal with the needs of applications. Thus, to be more efficient DIET is close to applications and can find the best resources allocation in accordance with their needs.

Chapter 9

DIET's Applications Scope

Contents

During this manuscript we have tried to give the most complete view of DIET as possible. All the researches lead around DIET have permitted a research prototype to become a middleware used in real environments. This means real applications can take benefit of DIET performances. Before concluding this manuscript, it seems relevant to briefly describe a certain number of applications (Geology, Aerospace, Robotic, Sparse solver, Cosmology, Climatology). This gives the application scope of DIET through real implementations. There are no dependencies between them, thus the order follow the chronological order of implementation. The last application corresponds to the Décrypthon project that is the production Grid platform of DIET.

9.1 Geology: Digital elevation model

Joint work with:

⋆ Sylvain Contassot-Vivier	: University Lumière Lyon 2. Lyon. France.
⋆ Frédéric Desprez	: INRIA. LIP Laboratory. Lyon. France.
⋆ Frédéric Lombard	: University of Franche-Comté. LIFC Laboratory. Besançon. France.
⋆ Jean-Marc Nicod	: University of Franche-Comté. LIFC Laboratory. Besançon. France.
⋆ Laurent Philippe	: University of Franche-Comté. LIFC Laboratory. Besançon. France.

This application results from a partnership between the LIP laboratory and the LST laboratory, both from ENS. The sequential version implemented by Michel Memier has been improved with a parallel version designed for Cluster platform [63]. The program computes a DEM (Digital Elevation Model). The input data are:

- Two satellite images of the same area, taken with two slightly different angles
- A set of additional parameters (e.g., camera localization, angle of the point of view, etc.)

The output is a 3D cloud of points graph representing a picture of the surface. The application consists in matching two points, respectively from both images and computes the space localization of others points of the picture. The resulting point cloud is mapped on a Grid where the heights are know and correspond to the DEM, as shown Figure 9.1.

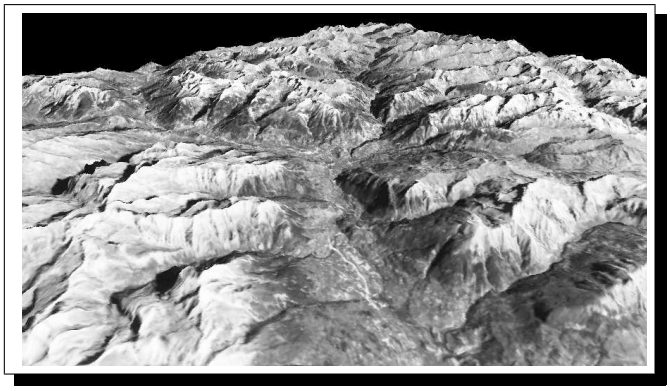


Figure 9.1: A textured 3D view of Digital Elevation Model.

This application is used to deal with the SPOT satellite pictures. The size of these images is 6000 × 6000 pixels, computing time is important and a parallel version is definitively relevant. The parallel version is based on the lines that fill the image. Each process receive a part of the image (a horizontal strip to compute and some lines bound come from above and below this area). Strips are distributed among processors. Experiments have show the benefit of the the parallel version from 2 to 125 processors. DIET was used to distribute each strip on available servers. The first web interface for DIET was designed to help the final user to use the application as can seen in Figure 9.2.

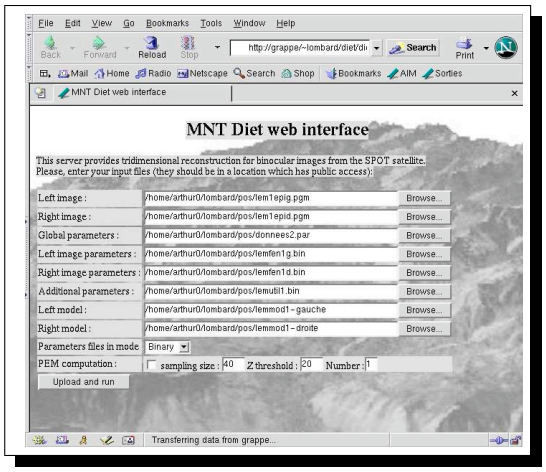


Figure 9.2: Web portal for the DEM application (MNT means DEM in french).

9.2 Aerospace: Finite element analysis

Joint work with:

⋆ Frédéric Desprez	: ENS Lyon. LIP Laboratory. Lyon. France.
⋆ Seung Jo Kim	: SNU. Department of Aerospace Engineering. Seoul. Korea.
⋆ Sihyoung Park	: SNU. Department of Aerospace Engineering. Seoul. Korea.
⋆ Christian Perez	: INRIA. IRISA. Rennes. France.
⋆ Antoine Vernois	: ENS Lyon. LIP Laboratory. Lyon. France.
⋆ Youngha Yoon	: SNU. Department of Aerospace Engineering. Seoul. Korea.

In this work, Ipsap, a massively parallel general-purpose finite element analysis program is incorporated for finite element analysis services (also called Mfem). Ipsap is based on the high performance implementation of a multifrontal solver modified for the finite element method [110]. There have been significant progress in the multifrontal methods and they are considered to be the most efficient direct solvers for the solution of general sparse system of linear equations. One of the most famous serial and parallel implementation is that of Gupta et al. [94] The implementation of common multifrontal methods deal with the global matrices in a given sparse format. The idea of the multifrontal method used in Ipsap is that the fronts matrix may be assembled in the merging stage even though the frontal matrix is reallocated in the re-ordered form [109]. It is algebraically equivalent to the generalized multifrontal methods, but uses the finite element assembling concept.

With the help of this idea, matrix assembling time can be significantly reduced. Also, the forming stage of the global sparse matrix is not required. In terms of communication overhead, such a merit is specially conspicuous and therefore Ipsap shows an apparent enhancement of performance in parallel environments. In Figure 9.3, the conceptual sketch of the multifrontal method is presented.

In addition to the multifrontal method, the block Lanczos iteration algorithm for generalized eigenvalue problems is implemented and incorporated in Ipsap. The block Lanczos iteration is also modified such that the memory structure allocated for the Lanczos basis is best suited for the multifrontal method. The main contribution of Ipsap eigenproblem solver lies in the sharing the interface DOF's enabling the mass matrix to be stored with each processor's scope without

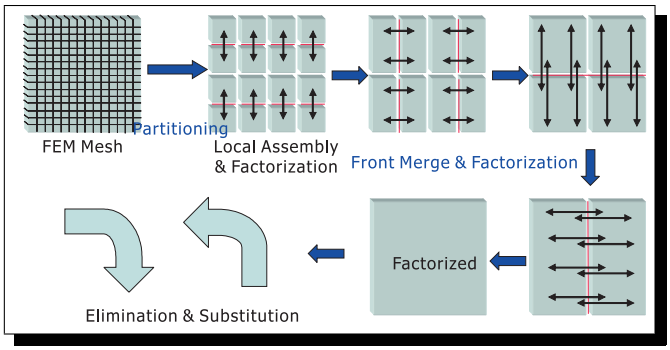
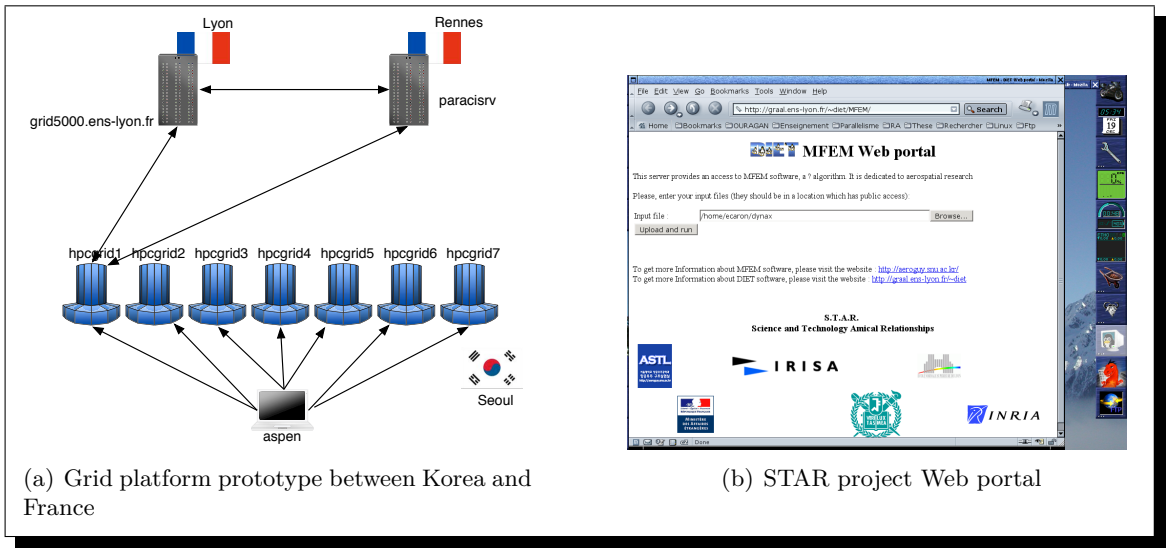


Figure 9.3: Modified multifrontal algorithm.

communications.

In conjunction with the STAR project, we have deployed this application on a large Korean cluster (with 500 CPUs and splitted in 7 cluster) and on several machines in France (Rennes and Lyon) as shown Figure 9.4(a). Moreover a web portal has been implemented also (see Figure 9.4(b))



(a) Grid platform prototype between Korea and France

(b) STAR project Web portal

Figure 9.4: Deployment of IPSAP using the MFEM service.

9.3 Robotic: Remote robot control

Work done by:

- ⋆ Fabrice Sabatier : SUPELEC. Metz. France
- ⋆ Amelia De Vivo : University di Salerno. Baronissi. Italy.
- ⋆ Stéphane Vialle : SUPELEC. Metz. France

Figure 9.5: Robot controlled through DIET.

This application is the first that interfaced with DIET by an external team. They focus on the complexity of remote control of autonomous robots. It is not as easy as it appears. It is not just a set of commands to send to a robot. Indeed, many questions must be solves before doing something, and generally, they need more processing power than available on the robots onboard processor.

As described in [148] "*the testbed application consists of three very frequently used basic modules: self-localization, navigation and lightness detection. Our robot navigates in dynamic environments, where no complete predetermined map can be used. Anyway, artificial landmarks are installed at known coordinates. When switched on, the robot makes a panoramic scan with its camera, detects landmarks and self-localizes. Based on its position, it can compute a theoretical trajectory to go somewhere. For error compensation, new self-localizations happen at intermediate positions. During navigation the robot checks the environment lightning and, eventually, signals problems. For this purpose it moves its camera and catches images for average lightness computation. The Koala Server always sends its clients JPEG compressed images. The three pilot modules were re-designed and turned into Grid services for robotic application development.*"

They designed a Grid architecture across Internet, including resources from two laboratories, one in Italy and one in France. It is based on the DIET GridRPC environment and supports distributed remote redundant control of an autonomous robot. Figure 9.6 shows the deployment phase.

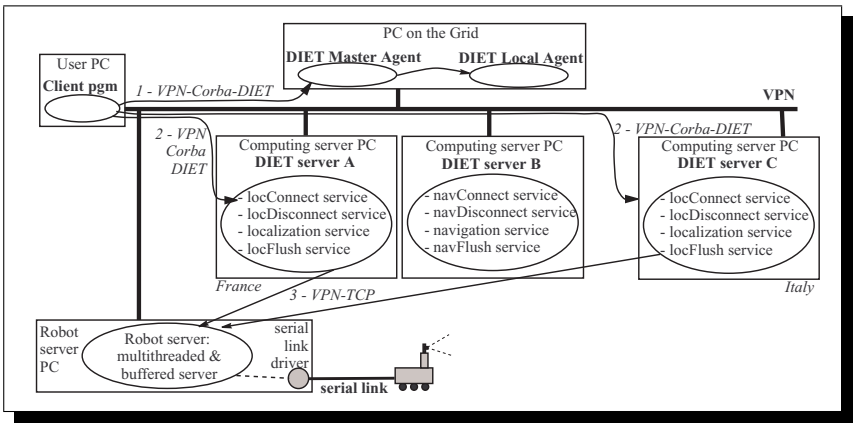


Figure 9.6: DIET deployment of the robotic application on Grid architecture

9.4 TLSE: Sparse linear system solvers

Joint work with:	
⋆ Michel Daydé	: IRIT Laboratory. Toulouse. France
⋆ Frédéric Desprez	: INRIA. LIP Laboratory. Lyon. France.
⋆ Christophe Hamerling	: CERFACS. Toulouse. France
⋆ Jean-Yves L'excellent	: INRIA. LIP Laboratory. Lyon. France.
⋆ Marc Pantel	: IRIT Laboratory. Toulouse. France
⋆ Chiara Puglisi	: IRIT Laboratory. Toulouse. France

Solving systems of linear equations is one of the key operations in linear algebra. Many different algorithms are available for that purpose. These algorithms require a very accurate tuning to minimize runtime and memory consumption. The TLSE project provides on one hand, a scenario-driven expert site to help users choose the right algorithm according to their problem and tune accurately this algorithm, and on the other hand, a test-bed for experts in order to compare algorithms and define scenarios for the expert site. Both features require to run the available solvers a large number of times with many different values for the control parameters (and maybe with many different architectures). Currently, only the Grid can provide enough computing power for this kind of application. The DIET middleware is the Grid backbone for TLSE (The Grid-TLSE project is a three-year project started in December 2002 and funded by the French Ministry of Research ACI GRID Program [13, 69]). It manages the solver services and their scheduling in a scalable way.

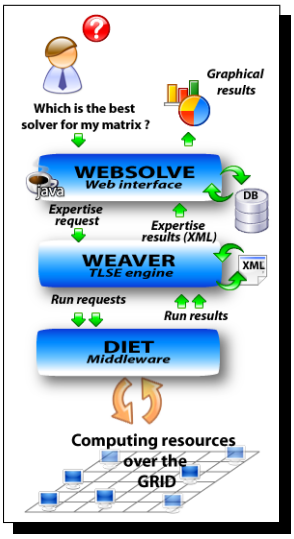


Figure 9.7: Architecture of the Grid-TLSE Project.

Three kinds of users may be interested in Grid-TLSE:

1. **End users**, with either basic, medium or advanced knowledge in numerical computations and parallel/distributed programming. They mainly want to choose the best solver for their problem according to specific metrics (memory usage, robustness, accuracy, execution time) and find out the control parameters' best values for the best solver.

2. **Experts** in numerical computation and parallel/distributed programming, who are involved in writing packages. Experts may want to compare solvers using sophisticated controls and metrics or add new solvers or new scenarios.

3. The Grid-TLSE **manager** who will take care of users, computers, and services, matrix collections, bibliography. The manager will also need to access the current state of the Grid and the list of available solvers.

The main components of the Grid-TLSE site are the followings. **WebSolve** allows a user using a standard web navigator to submit requests for computation or expertise to a Grid, browse the matrix database, upload/download a matrix, monitor the submitted requests, manage and add solvers and scenarios, and finally check for their correct execution. Most of the Web interface is dynamic: it is built according to the meta-data. **Weaver** converts a general request for expertise into sequences of elementary solver runs.

It is also in charge of the deployment and the exploitation of services over a Grid through the DIET middleware. The expertise providing kernel is fully dynamic in the same sense as WebSolve, all the services rely on the meta-data. DIET provides an access to solvers and data. Finally, the **Database** stores the required data for the whole project. In particular, it contains all the meta-data.

First, the user interacts with the WebSolve interface in order to choose an expertise scenario (the objective of the session) and provide the appropriate parameters for this scenario. Then, this request is forwarded to the Weaver kernel. According to the description of the scenario, Weaver builds one or more expertise steps (which correspond to execution operators in the scenarios). Each expertise step produces an experiment plan. An experiment is a partially valued set of features which represents a solver run. Running an experiment will forward this set to the appropriate solver on the Grid thought DIET, which will send back the fully valued experiment resulting from the solvers run. All the results of an experiment plan are processed according to the scenario in order to produce the next expertise step. And finally, the results of the last experiment plan are forwarded to WebSolve, which stores all the raw results and then produces synthetic graphics according to the scenario and the user's request.

The number of expertise steps is therefore dynamic and depends on the results of the experiments. The expertise process terminates as the only iterative operators in a scenario are, on one hand a *foreach* applied to finite sets of values, and on the other hand, recursive traversal of finite static trees used in the meta-data. The scenario is therefore a kind of dynamic workflow whose execution depends on the intermediate results.

DIET is used in order, on one hand, to schedule and execute experiments on the most adapted available solvers on the Grid, and on the other hand, to share intermediate results inside an experiment plan or between various experiment plans. Scenarios express data-flow dependencies which are used by the DIET persistence facilities in order to reduce the communication costs by an appropriate scheduling.

9.5 Cosmology: Formation of galaxies

Joint work with:

- ⋆ Jeremy Blaizot : ENS Lyon. CRAL. Lyon. France.
- ⋆ Hélène Courtois : ENS Lyon. CRAL. Lyon. France.
- ⋆ Benjamin Depardon : ENS Lyon. LIP Laboratory. Lyon. France.
- ⋆ Romain Teyssier : CEA. Saclay. France.

RAMSES is a typical computational intensive application used by astrophysicists to study the formation of galaxies. RAMSES is used, among other things, to simulate the evolution of a collisionless, self-gravitating fluid called "dark matter" through cosmic time. Individual trajectories of macro-particles are integrated using a state-of-the-art "N body solver", coupled to a finite volume Euler solver, based on the Adaptive Mesh Refinement techniques. The computational space is decomposed among the available processors using a mesh partitioning strategy based on the Peano-Hilbert cell ordering. Cosmological simulations are usually divided into two main categories. Large scale periodic boxes requiring massively parallel computers are performed on very long elapsed time (usually several months). The second category stands for much faster small scale "zoom simulations". One of the particularity of the HORIZON project is that it allows the re-simulation of some areas of interest for astronomers (Figure 9.8).

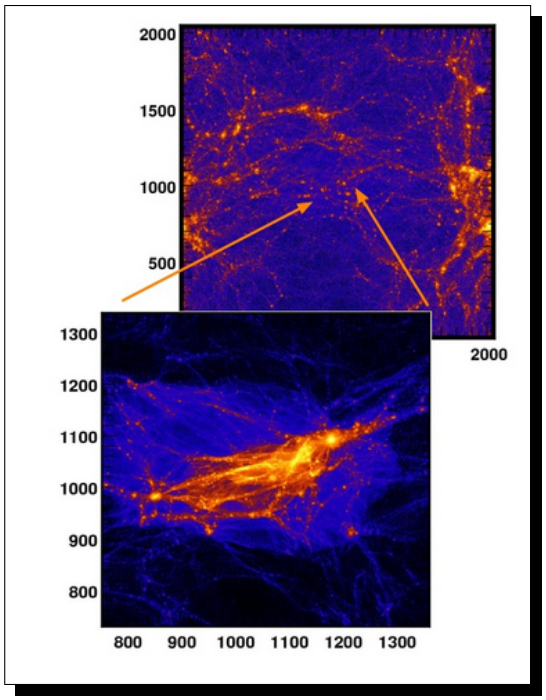


Figure 9.8: Re-simulation on a supercluster of galaxies to increase the resolution

Performing a zoom simulation requires two steps: the first step consists of using RAMSES on a low resolution set of initial conditions (i.e., with a small number of particles) to obtain at the end of the simulation a catalog of "dark matter halos", seen in Figure 9.9 as high-density peaks, containing each halo position, mass and velocity.

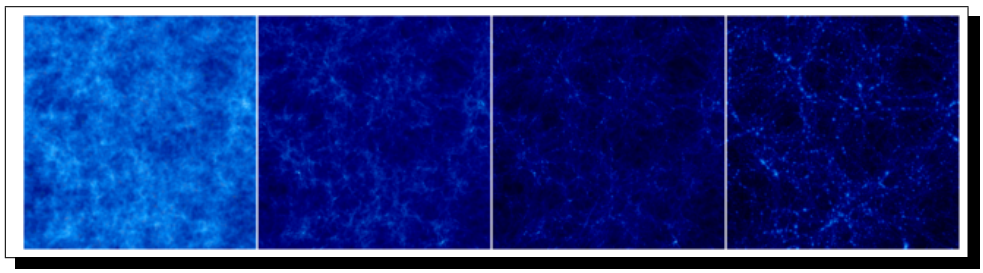


Figure 9.9: Time sequence (from left to right) of the projected density field in a cosmological simulation (large scale periodic box).

A small region is selected around each halo of the catalog, for which we can start the second step of the "zoom" method. The idea is to resimulate this specific halo at a much better resolution. For that, we add in the Lagrangian volume of the chosen halo a lot more particles, in order to obtain more accurate results. Similar "zoom simulations" are performed in parallel for each entry of the halo catalog and represent the main resource consuming part of the project.

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

We have deployed DIET on 5 sites on Grid'5000 (6 clusters). We studied the possibility of computing a lot of low-resolution simulations. The client requests a 1283 particles 100Mpc.h-1 simulation (first part). When it receives the results, it requests simultaneously 100 sub-simulations (second part). As each server cannot compute more than one simulation at the same time, we won't be able to have more than 11 parallel computations at the same time. The experiment (including both the first and the second part of the simulation) lasted 16h 18min 43s (1h 15min 11s for the first part and an average of 1h 24min 1s for the second part). The benefit of running the simulation in parallel on different clusters is clearly visible: it would take more than 141h to run the 101 simulation sequentially.

9.6 Climatology: Ocean-Atmosphere modelization

Joint work with:

⋆ Yves Caniou	: University of Lyon Claude Bernard. LIP Laboratory. Lyon. France.
⋆ Ghislain Charrier	: ENS Lyon. LIP Laboratory. Lyon. France.
⋆ Andreea Chis	: ENS Lyon. LIP Laboratory. Lyon. France.
⋆ Frédéric Desprez	: INRIA. LIP Laboratory. Lyon. France.
⋆ Eric Maisonnave	: CERFACS. Toulouse. France.

World's climate is currently changing due to the increase of the greenhouse gases in the atmosphere. Climate fluctuations are forecasted for the years to come. For a proper study of the incoming changes, numerical simulations are needed, using general circulation models of a climate system (atmosphere, ocean, continental surfaces) on forced mode or coupled mode (i.e., allowing information exchanges between each component during simulation). Imperfection of the models and global insufficiency of observations make it difficult to tune model parametrization with precision. Uncertainty on climate response to greenhouse gases can be investigated by performing an ensemble prediction with varying parameters. Climatologists' strategy, in our case, is to launch parallel simulations. Each independent simulation models the evolution of the present climate followed by the 21^{st} century. All simulations have a distinct physical parametrization of clouds dynamics, which primacy in such studies has been emphasized in [112]. Comparing independent simulations, they expect to better understand the relations between the variation in this parametrization with the variation in climate sensitivity to greenhouse gases. We have analyzed and modelized the application in [38] with the purpose of deriving appropriate scheduling heuristics in order to decrease the execution time of such applications.

An experiment is composed of several 1D meshes of identical DAGs composed of parallel tasks. To obtain a good completion time, we divide groups of processors into sets each working on parallel tasks. Thus, the group sizes are chosen by computing the best makespan for several grouping possibilities. We improved this heuristic method by different means. The improvement yielding to the best makespan is the representation of the problem as an instance of the Knapsack problem. As this heuristic is firstly designed for homogeneous platforms, in [38] we present its adaptation to heterogeneous platforms. Simulations show improvements of the makespan up to 12%.

9.7 Bioinformatics: BLAST

Joint work with:
- ⋆ Vincent Breton : CNRS. IN2P3. Clermont-Ferrand. France
- ⋆ Gaël Le Mahec : CNRS. IN2P3. Clermont-Ferrand. France
- ⋆ Frédéric Desprez : INRIA. LIP Laboratory. Lyon. France.

Genomics involving BLAST (Basic Local Alignment Search Tool) applications [11, 113] are the common tool for searching sequences similarities within protein and DNA databases. BLAST is used to search homologies between amino-acids or nucleotides sequences. It quantifies similarities using some sensibility parameters and a similarity score matrix. Sequences are known as similar if they have enough common characters and not too many differences which can be substitutions, deletions or insertions. The search for similarities between sequences can be done at a *global* or *local* level. At the global level, we search for the best alignment between two sequences permitting large different area inside the sequences [132]. At the local level, we search subsequences alignments, not using the subsequences without any alignment for the total similarity score computing [155, 87].

However, as the size of available databases [1] increases (now up to hundreds of Gigabytes), the search of many sequences is time consuming. Many researches have been done to get faster algorithms. Their goal is to reduce the search scope while looking at high scoring alignments. Heuristics have been produced and among them BLAST is the most popular. Still, as databases' size and number still increase, the use of parallel computing becomes mandatory. Several approaches exist from parallelizing the BLAST kernel itself (and distributing the databases) to splitting the input sequences to different servers and replicating the databases. All these approaches can be combined to get the best performance.

The DIET BLAST [34] application was designed to manage thousands of BLAST requests over large biological databases. We tested it, submitting 40000 requests over 5 databases of different sizes (from 1 to 5 GB). Each request is submitted asynchronously to the platform, DIET selects the computation node using a scheduling algorithm. We tested four different scheduling policies: a simple greedy algorithm, MCT, SRA and dynamic-SRA. A gains of 30% on completion time using the dynamic-SRA algorithm has been reach.

9.8 Bioinformatics: The Décrypthon Grid for neuromuscular discorder

Joint work with:
- ⋆Nicolas Bard : CNRS. LIP Laboratory. Lyon. France.
- ⋆Raphaël Bolze : CNRS. LIP Laboratory. Lyon. France.
- ⋆ Frédéric Desprez : INRIA. LIP Laboratory. Lyon. France.
- ⋆Anne Friedrich : IGBMC, France.
- ⋆Luc Moulinier : IGBMC, France.
- ⋆Ngoc-Hoan Nguyen : IGBMC, France.
- ⋆Michaël Heymann : CNRS. LIP Laboratory. Lyon. France.
- ⋆ Thierry Toursel : AFM, France.

The rapid progress of biotechnologies and information technologies has generated numerous large biomedical datasets. Interpretation and exploitation of these heterogeneous high-throughput datasets require increasingly complex information management and analysis. In this context, the

[1] `http://www.ncbi.nih.gov/`

Décrypthon program [2], a Grid technology platform was launched in 2005 by the AFM [3] (French Muscular Dystrophy Association) and IBM, historic partners since 2001, with the collaboration of the CNRS (French National Center for Scientific Research) and French universities (Bordeaux 1, Lille 1, Paris 6 Jussieu, ENS Lyon, Orsay, CRIHAN in Rouen). The Décrypthon Grid aims to promote access to computational resources to foster the development of large-scale research programs in the field of biology, and in particular in the context of Neuromuscular Disorders (ND). The close partnership of the AFM and IBM also led to the launch of the *Help Cure Muscular Dystrophy* on the *World Community Grid* [4]. Following the success of the first phase [28], the second phase of this project is dedicated to the study of protein-protein interactions for more than 2000 gene candidates involved in ND[5] with the final objective of predicting whether two proteins are potential interacting partners in the cell. In February 2010, the project involves two major programs (the Joint Evolutionary Trees (JET) program and the docking algorithm MAXDo) and the Grid computing time available for this project is equivalent to 50 years CPU per day.

9.8.1 The Décrypthon platform

The Décrypthon resources managed by DIET are constituted of multiprocessor machines under the AIX operating system, and a cluster of single processor machines under the linux system. The machines of the Grid are very heterogeneous and managed by two different batch schedulers (OAR and Loadleveler). This is a total of 58 machines for 475 processors, however only about 40 percent of the Grid is available for the Décrypthon users: the resources are shared with the local users of the universities. The Décrypthon Grid also has two web servers on the Orsay site. Figure 9.10 presents the overall architecture of the platform.

The Décrypthon Grid is monitored by a java/JSP application called DIET WebBoard introduced Section 2.8.6. The architecture of the WebBoard is comprised of **DIET WebBoard Admin**, a web application to administrate the Grid, and the **GoDIET** (see Section 5.4) the tool to deploy the DIET platform. Each scientific application on the Grid has a **dedicated Web Interface** for the users. The web applications update in real time the WebBoard database (Postgres or MySQL), containing the status of each element of the architecture (Storage space, Jobs, Results, Working nodes, etc.). The WebBoard uses a generic DIET client that can submit a job for any of the applications.

Six major functions are implemented in the DIET WebBoard. Using the **GoDIET** tool, the platform can be launched, stopped or checked for its integrity. The concept of **Jobs** and **Workunits** is added as an abstraction of the DIET tasks in order to exploit data parallelism on the servers. Users are **identified** using a login and a password and actions on the interfaces are restricted. It is also possible to run the DIET WebBoard under the https protocol. Jobs are monitored and a job that is enable to finish before its walltime expires (for example when a cluster is shut down) can be copied. **Statistics** on the usage of the Grid are available (daily Gantt charts, CPU time used each day of a month, repartition of the load between the sites, and user defined charts). Many other minor functions are also implemented such as sending mail to users when jobs are completed, managing of the local and remote storage spaces for results and data, dumping the database, an RSS feed.

[2] `http://www.decrypthon.fr/english/`
[3] AFM: `http://www.afm-france.org/afm-english_version/`
[4] WCG: `http://www.worldcommunitygrid.org/`
[5] `http://www.ihes.fr/~carbone/HCMDproject.htm`

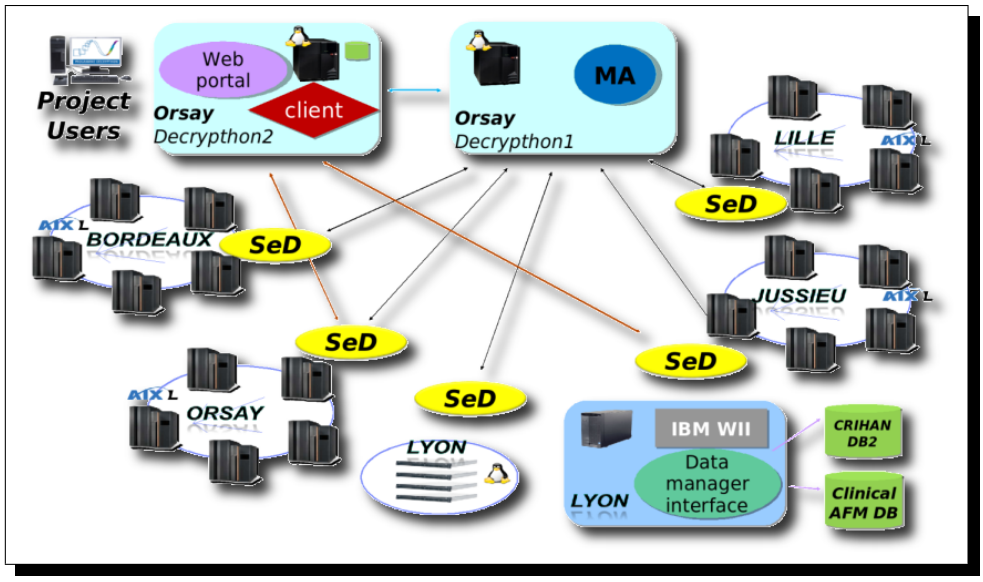


Figure 9.10: The architecture of the Décrypthon Grid.

9.8.2 The SM2PH application

A number of biological projects that require significant computing power are now taking advantage of the technological capabilities provided by the Décrypthon computing Grid.? Among these, the SM2PH project [3] aims to develop an infrastructure dedicated to the understanding of the correlation between gene variations and the associated human genetic diseases, which represents a major challenge in the post-genomic era. As a first approximation, this so-called "genotype-phenotype relationship" can be addressed by characterizing how genetic alterations affect gene products (proteins) at the molecular level. In this context, SM2PH considers the phenotypes associated with human pathologies by defining the structural, functional and evolutionary context of all the gene/protein known to be involved in human diseases. At the computer level, the establishment of the SM2PH infrastructure required the interoperability of numerous programs (more

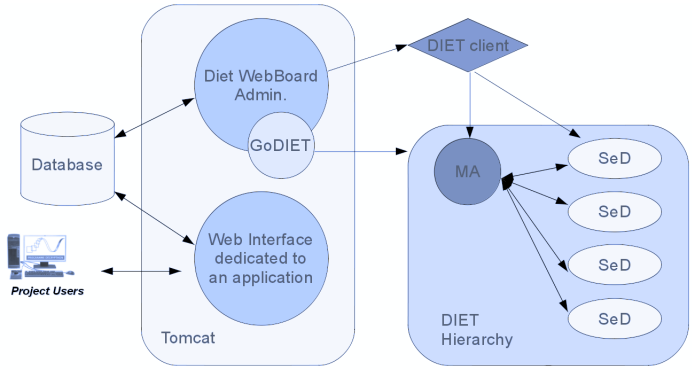


Figure 9.11: The architecture of DIET WebBoard.

than 20) and the development of automatic procedures to compute and integrate information from heterogeneous sources as well as to perform regular updates.

A web interface dedicated to the SM2PH application has been developed using the DIET WebBoard, which allows identified users to submit their input data (files containing thousands of sequences) and to select specific parameters. The upload of the user file initiates a job, with one workunit for each sequence in the submitted file. A DIET client submits these workunits to the Grid and the selected DIET SeD (on the least loaded cluster) submits a batch scheduler script. Finally, the SM2PH main application is launched on the worker node. The Décrypthon web interface monitors the data sent by the users and surveys the DIET jobs. The state of each workunit can be viewed (waiting, running, completed) and if the result has not been received when the walltime expires, the workunit is re-submitted. The final output of the application is a set of 7 files per submitted sequence. All these results are automatically gathered by the Grid server, and up to 5000 results can be downloaded at once by the user.

In February 2010, 2362 human proteins are known to be involved in human monogenic diseases. These proteins represent the input dataset of the SM2PH project. A database named SM2PH-db [3] has been developed to provide access to the scientific community to various querying and analysis tools and to the retrieved or predicted data, together with a wealth of interconnected information for each input gene/protein. To guarantee the sustainability of the SM2PH-db, the SM2PH application is automatically launched on the Grid every 2 months, ensuring that the user is working with up-to-date information. The advantages of the Décrypthon Grid in this context is clear, since the complete update takes only 1 days instead of the estimated 15 days if it was performed locally.

9.9 Conclusion and future works

In this Chapter we have shown through several examples that DIET can deal with a lot of different application. Indeed, the process to use all these applications on a Grid is always the same: we study the applications needs, and sometimes even their internal structure, the DIET is tuned to be efficient with the application.

Applications have driven DIET from a prototype for scheduling algorithms experimental validation, to a middleware used in a Grid production platform. Obviously, a future work related to this chapter is to increase the range of applications. Moreover, closer to the integration of applications point of view, new developments are in progress to provide a GUI integrated in the Eclipse framework, to help users with their client and server programs.

Hence, this is no wonder we chose “Adapt the world to your application” as a catchphrase for the commercial version of DIET. This shows the philosophy behind all the developments that lead to what DIET is nowadays.

Conclusion

This manuscript gives a complete overview of the Grid and Cloud middleware DIET. The tree DIET architecture is described and three GridRPC components, client, scheduler and server are discussed from the point of view of DIET. We also presented clues to understand how to use DIET. A brief introduction around the six associated tools in the sphere of incluence of DIET are given. Some researches have been lead around service discovery, when the hierarchy of DIET grows to an aggregation of trees. We focused on resources management within DIET. We showed how DIET deals with Local Resources Management Systems and Cloud platform. Some researches are driven by the deployment and planing steps. We have designed a modelization to help DIET platform administrators find the best (on homogeneous platform) or an efficient (on heterogeneous platform) deployment of this distributed middleware. After the deployment and the service discovery, the first problem to address concerns data management. We introduced three solutions implemented in DIET to give an answer to this question. The following sections dealt with the management of computing tasks. The workflow management available in DIET is described. This way users can submit a set of requests (mainly useful in case of data dependencies between them) and thus benefit of the scheduling made by our middleware. Finally as a pre-conclusion, we have chosen the most significant applications that took benefit of DIET, and provides a Grid version of them. DIET was downloaded 800 times over 42 country since 2005. The success story of DIET relies on two main events. The first one was when DIET entered in production to serve the Décrypthon project. The second one was when DIET has been involved in a start'up. A second life begins for DIET. One of the first evolution is to increase the code quality through a better modularity using Boost [8].

During these years, as the technical chief of the development of DIET, I have been watchful to different points:

Adaptability: a research software must be open to any research area or concept and collaborations. A research software should always offer the bridges required to increase the relationship between software.

Modularity: many functionalities of DIET can be switched on or off at the compilation step or during run time. Indeed, DIET can be customized to fit the needs of the application and the targeted platform.

Scalability: one of the main advantage of DIET is high scalability. Thus, I prohibited all centralized solutions, avoided any bottleneck (except for the final display of the visualization part of course). Even if that was a simpler way to implement a functionality. For instance, scheduling is done using a distributed approach that increases the complexity to be close to the optimal solution, but implies some challenges to solve.

Stability: In an academic research project, many students, engineers, collaborators[6] are involved in a project. That implies many personal backgrounds, many ways of developing. Nevertheless, with the use of external collaborative tools for the development (i.e., CVS, Mailing List, Bugzilla[7], CDash[8], etc.) and the quality of relationship between the DIET developers, we have preserved an high stability during the development process.

User Friendly Interface: a key of the success of DIET was the easiness for a user to transform his application to a grid application. Thus we kept in mind that the large available functionalities of DIET must be hide behind an API as simple as possible. Moreover, I required the development of a Graphic User Interface which has been very attractive application users and very fruitful for demonstrations (DIET was in demonstration in 6 SuperComputing 2002, 2004, 2005, 2007, 2008 and 2009 and will be attending to the next one).

As a research point of view, many projects are related to DIET. Among them, we plan to extend **DIET Cloud**. As we have seen in Section 4.3, DIET provides a solution to address a Cloud platform through Eucalyptus [134]. However, DIET is a middleware and could be used to hide the Cloud heterogeneity as it hides it for the Grid. Thus DIET can add different Cloud solution (e.g., OpenNebula [157] or Nimbus [9]) interfaces. Hence DIET can increase and provide a good scheduling for the Sky Computing paradigm [105]. We have already started a work to provide forecast the resources reservation behavior of the Cloud platform [51].

Reducing power consumption is a big challenge that already motivated a lot of researches in self-sufficient mobile systems. Computer centers turned into having more and more servers, and thus consume a lot of electrical power. Thus under the code name **DIET Green** we want to reduce energy by offering a green scheduler for the grid middleware DIET.

As has been discussed as a conclusion of Chapter 4, with the **DIET SMP** project we plan to provide an access to virtual SMP resources through Kerrighed. This software virtualizes a cluster to an SMP (see Section 4.4).

We want to extend the planning algorithms for the deployment (introduced Chapter 5) to deal with the dynamicity of DIET (adding and removing of DIET components) and the load of the platform during runtime. Moreover, the merge between research and the deployment software should be finalized.

We would also like to ...

... Stop! Please! As usual when I talk about DIET, I consume too many time, too many pages. Then, it's time for me to close this manuscript and to turn the page on the first part of the life of DIET. It was a fabulous adventure for so many reasons. No matter how, a very special thanks you to each person involved in this story.

[6] Complete list is available here: `http://graal.ens-lyon.fr/DIET/team-list.html`
[7] `http://graal.ens-lyon.fr/bugzilla/`
[8] `http://cdash.inria.fr/CDash/index.php?project=DIET`
[9] `http://www.nimbusproject.org`

List of Figures

Bibliography

[1] Optimal Blade System Design of a New Concept VTOL Vehicle Using the Departmental Computing Grid System. In *ACM/IEEE Super Computing 2004 Conference (SC'04)*, page 36. Pittsburgh, PA. USA., 2004.

[2] Amazon Elastic Compute Cloud, 2008.

[3] SM2PH-db: an interactive system for the integrated analysis of phenotypic consequences of missense mutations in proteins involved in human genetic diseases. *Hum Mutat.*, V31 Issue 2:127–135, 2009.

[4] GNU Linear Programming Toolkit, 2010. `http://www.gnu.org/software/glpk`.

[5] ILOG CPLEX, 2010. `http://www-01.ibm.com/software/integration/optimization/cplex`.

[6] lplsolve 5.5, 2010. `http://lpsolve.sourceforge.net/5.5`.

[7] I. Abdul-Fatah and S. Majumdar. Performance of CORBA-Based Client-Server Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 13:111–127, 2002.

[8] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[9] E. Alba, F. Almeida, M. Blesa, J. Cabeza, C. Cotta, M. Díaz, I. Dorta, J. Gabarró, C. León, J. Luna, L. Moreno, C. Pablos, J. Petit, A. Rojas, and F. Xhafa. A Library of Skeletons for Combinatorial Optimisation. *Euro-Par 2002 Parallel Processing*, pages 63–73, 2002.

[10] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster. The Globus striped GridFTP framework and server. In *Proceedings of the ACM/IEEE Symposium on Supercomputing (SC'05)*. IEEE Computer Society Press, 2005.

[11] S. Altschul, W. Gish, E. Miller, E. Myers, and D. Lipman. A Basic Local Alignment Search Tool. *J. Mol. Biol.*, 215:403–410, 1990.

[12] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J Mol Biol*, 215(3):403–410, October 1990.

[13] P. Amestoy and M. Pantel. Grid-TLSE: A Web expertise site for sparse linear algebra. In *Sparse Days and Grid Computing in St Girons*. june 2003. Http://www.cerfacs.fr/algor/PastWorkshops/SparseDays2003.

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

[15] G. Antoniu, M. Bertier, L. Bougé, E. Caron, F. Desprez, M. Jan, S. Monnet, and P. Sens. GDS: An Architecture Proposal for a Grid Data-Sharing Service. In V. Getov, D. Laforenza, and A. Reinefeld, editors, *Future Generation Grids*, volume XVIII, CoreGrid Series of *Proceedings of the Workshop on Future Generation Grids November 1-5, 2004, Dagstuhl, Germany*. Springer Verlag, 2006.

[16] G. Antoniu, L. Bougé, and M. Jan. JuxMem: An adaptive supportive platform for data sharing on the grid. *Scalable Computing: Practice and Experience*, 6(33):45–55, 2005.

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

[18] M. Arenas, P. Collet, A. Eiben, M. Jelasity, J. Merelo, B. Paechter, M. Preuß, and M. Schoenauer. A Framework for Distributed Evolutionary Algorithms. *Parallel Problem Solving from Nature —PPSN VII*, pages 665–675, 2002.

[19] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[20] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4. UTK Computer Science Dept. Technical Report CS-01-467, University of Tennessee, Knoxville, TN, July 2001. `http://www.cs.utk.edu/netsolve/`.

[21] J. Aspnes and G. Shah. Skip Graphs. In *Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. January 2003.

[22] N. Bard, R. Bolze, E. Caron, F. Desprez, M. Heymann, A. Friedrich, L. Moulinier, N.-H. Nguyen, O. Poch, and T. Toursel. Décrypthon Grid - Grid Resources Dedicated to Neuromuscular Disorders. In *The 8th HealthGrid conference*. Paris, France, June 2010. To appear.

[23] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177. ACM, New York, NY, USA, 2003. ISBN 1-58113-757-5.

[24] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *SOSP'03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177. ACM, New York, NY, USA, 2003. ISBN 1-58113-757-5.

[25] D. Bein, D. A.K., and V. V. Snap-Stabilizing Optimal Binary Search Tree. In S. LNCS 3764, editor, *Proceedings of the 7th International Symposium on Self-Stabilizing Systems (SSS '05)*, pages 1–17. 2005.

[26] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *ATEC'05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41. USENIX Association, Berkeley, CA, USA, 2005.

[27] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski. The GrADS Project: Software Support for High-Level Grid Application Development. *The International Journal of High Performance Computing Applications*, 15(4):327–344, November 2001.

[28] V. Bertis, R. Bolze, F. Desprez, and K. Reed. From Dedicated Grid to Volunteer Grid: Large Scale Execution of a Bioinformatics Application. *Journal of Grid Computing*, 7(4):463–478, December 2009.

[29] A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *Proceedings of the SIGCOMM Symposium*. August 2004.

[30] R. Bolze. *Analyse et déploiement de solutions algorithmiques et logicielles pour des applications bioinformatiques à grande échelle sur la grille*. Ph.D. thesis, École Normale Supérieure de Lyon, October 2008.

[31] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and T. Irena. Grid'5000: a large scale and highly reconfigurable experimental Grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, November 2006.

[32] 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, editor, *Computational Science and Its Applications - ICCSA 2006*, volume 3984 of *LNCS*, pages 202–213. Springer, Glasgow, UK., May 8-11 2006. ISBN 3-540-34079-3.

[33] R. Bolze, F. Desprez, and B. Isnard. Evaluation of Online Multi-Workflow Heuristics based on List-Scheduling Algorithms. Gwendia report L3.3.

[34] V. Breton, E. Caron, F. Desprez, and G. Le Mahec. *Handbook of Research on Computational Grid Technologies for Life Sciences, Biomedicine and Healthcare*, chapter High Performance BLAST Over the Grid. IGI Global, 2009.

[35] L. Broto, D. Hagimont, P. Stolf, N. Depalma, and S. Temate. Autonomic Management Policy Specification in TUNe. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 1658–1663. ACM, New York, NY, USA, 2008. ISBN 978-1-59593-753-7.

[36] A. Buble, L. Bulej, and P. Tuma. CORBA Benchmarking: A Course with Hidden Obstacles. In *IPDPS'03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 279.1. IEEE Computer Society, Washington, DC, USA, 2003. ISBN 0-7695-1926-1.

[37] S. Cahon, N. Melab, and E. G. Talbi. ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics. *Journal of Heuristics*, 10(3):357–380, 05 2004.

[38] Y. Caniou, E. Caron, G. Charrier, A. Chis, F. Desprez, and E. Maisonnave. Ocean-Atmosphere Modelization over the Grid. In W.-c. Feng and Y. Yang, editors, *The 37th International Conference on Parallel Processing (ICPP 2008)*, pages 206–213. IEEE, Portland, Oregon. USA, September 2008.

[39] Y. Caniou, E. Caron, H. Courtois, B. Depardon, and R. Teyssier. Cosmological Simulations using Grid Middleware. In *Fourth High-Performance Grid Computing Workshop (HPGC'07)*. IEEE, Long Beach, California, USA, March 2007.

[40] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron, and O. Richard. A batch scheduler with high level components. In *Cluster computing and Grid 2005 (CCGrid05)*. 2005.

[41] 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. IEEE/ACM, Seattle, USA, November 2005.

[42] E. Caron, S. Chaumette, S. Contassot-Vivier, F. Desprez, E. Fleury, C. Gomez, M. Goursat, E. Jeannot, D. Lazure, F. Lombard, J.-M. Nicod, L. Philippe, M. Quinson, P. Ramet, J. Roman, F. Rubi, S. Steer, F. Suter, and G. Utard. Scilab to Scilab//, the OURAGAN Project. *Parallel Computing*, 11(27):1497–1519, October 2001.

[43] E. Caron, P. K. 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 Applications and Tools. In conjunction with HPDC-15.*, pages 1–8. Paris, France, June 2006.

[44] E. Caron, P. K. Chouhan, and A. Legrand. Automatic Deployment for Hierarchical Network Enabled Server. In *The 13th Heterogeneous Computing Workshop (HCW 2004)*, page 109b (10 pages). Santa Fe. New Mexico, April 2004.

[45] E. Caron, A. Datta, F. Petit, and C. Tedeschi. Self-stabilization in tree-structured P2P Service Discovery Systems. In *27th International Symposium on Reliable Distributed Systems (SRDS 2008)*, pages 207–216. IEEE, Napoli, Italy, October 2008.

[46] E. Caron, B. Depardon, and F. Desprez. Deployment of a hierarchical middleware. In *Euro-Par 2010*. Institute for High Performance Computing and Networking of the Italian National Research Council, Ischia, Italy, August 31 to September 3. 2010. To appear.

[47] E. Caron and F. 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.

[48] E. Caron, F. Desprez, C. Fourdrignier, F. Petit, and C. Tedeschi. A Repair Mechanism for Fault-Tolerance for Tree-Structured Peer-to-Peer Systems. In Y. Robert, M. Parashar, R. Badrinath, and V. K. Prasanna, editors, *HiPC'2006. 13th International Conference on High Performance Computing.*, volume 4297 of *LNCS*, pages 171–182. Springer-Verlag Berlin Heidelberg, Bangalore. India, December 18-21 2006.

[49] E. Caron, F. Desprez, and G. Le Mahec. Parallelization and Distribution Strategies of Large Bioinformatics Requests over the Grid. In S. B. . Heidelberg, editor, *Algorithms and Architectures for Parallel Processing*, volume Volume 5022/2008 of *Lecture Notes in Computer Science*, pages 257–260. 2008.

[50] E. Caron, F. Desprez, J.-Y. L'Excellent, C. Hamerling, M. Pantel, and C. Puglisi-Amestoy. Use of A Network Enabled Server System for a Sparse Linear Algebra Application. In V. Getov, D. Laforenza, and A. Reinefeld, editors, *Future Generation Grids*, volume XVIII, CoreGrid Series of *Proceedings of the Workshop on Future Generation Grids November 1-5, 2004, Dagstuhl, Germany*. Springer Verlag, 2006.

[51] E. Caron, F. Desprez, and A. Muresan. Forecasting for Cloud computing on-demand resources based on pattern matching. Research Report RR-7217, Institut National de Recherche en Informatique et en Automatique (INRIA), http://hal.archives-ouvertes.fr/docs/00/49/69/89/PDF/RR-7217.pdf, July 2010.

[52] E. Caron, F. Desprez, F. Petit, and C. Tedeschi. Snap-stabilizing Prefix Tree for Peer-to-peer Systems. In *9th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, volume 4838 of *Lecture Notes in Computer Science*, pages 82–96. Springer Verlag Berlin Heidelberg, Paris, France, November 2007.

[53] E. Caron, F. Desprez, and F. Suter. Parallel Extension of a Dynamic Performance Forecasting Tool. *Parallel and Distributed Computing Practice (PDCP)*, 6(1):57–69, March 2003. Special Isue: Internet-Based Computing.

[54] E. Caron, F. Desprez, and C. Tedeschi. A Dynamic Prefix Tree for the Service Discovery Within Large Scale Grids. In A. Montresor, A. Wierzbicki, and N. Shahmehri, editors, *The Sixth IEEE International Conference on Peer-to-Peer Computing, P2P2006*, pages 106–113. IEEE, Cambridge, UK., September 2006.

[55] E. Caron, F. Desprez, and C. Tedeschi. Efficiency of Tree-structured Peer-to-peer Service Discovery Systems. In *Fifth International Workshop on Hot Topics in Peer-to-Peer Systems (Hot-P2P)*. In conjunction with IPDPS 2008, Miami, Florida, April 2008.

[56] H. Casanova and F. Berman. *Grid Computing*, chapter Parameter Sweeps on the Grid with APST. Wiley Series in Communications Networking & Distributed Systems. Wiley, 2003.

[57] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *Heterogeneous Computing Workshop*, pages 349–363. 2000.

[58] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. *Scientific Programming*, 8(3):111–126, 2000.

[59] P. Chan and D. Abramson. A Scalable and Efficient Prefix-Based Lookup Mechanism for Large-Scale Grids. In *3rd IEEE International Conference on e-Science and Grid Computing (e-Science 2007)*, pages 352–359. IEEE, Bangalore, India, December, 10-13 2007.

[60] E. Chang. Echo Algorithms: Depth Parallel Operations on General Graphs. *IEEE Trans. on Software Engineering*, SE-8:391–401, 1982.

[61] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The Data Grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23(3):187–200, 2000.

[62] P. K. Chouhan. *Automatic Deployment for Application Service Provider Environments*. Thesis, École Normale Supérieure de Lyon, September28 2006.

[63] S. Contassot-Vivier and S. Miguet. Parallel Visualization of Texture-Mapped Digital Elevation Models. In *IWPIA'95 International Workshop on Parallel Image Analysis*, pages 269–282. Lyon, France, December 1995.

[64] L. Cudennec, G. Antoniu, and L. Bouge. CORDAGE: Towards Transparent Management of Interactions Between Applications and Ressources. In *STHEC/ICS 2008*. Island of Kos, Aegean Sea, Greece, June 2008.

[65] H. Dail and F. Desprez. Experiences with Hierarchical Request Flow Management for Network Enabled Server Environments. Technical Report TR-2005-07, LIP ENS Lyon, 2005.

[66] H. Dail, O. Sievert, F. Berman, H. Casanova, S. YarKahn, J. Dongarra, C. Liu, L. Yang, D. Angulo, and I. Foster. *Grid Resource Management*, chapter Scheduling in the Grid Application Development Software Project, pages 73–98. Kluwer Academic Publisher, September 2003.

[67] S. Dandamudi and S. Ayachi. Performance of Hierarchical Processor Scheduling in Shared-Memory Multiprocessor Systems. *IEEE Trans. on Computers*, 48(11):1202–1213, 1999.

[68] A. Datta, M. Hauswirth, R. John, R. Schmidt, and K. Aberer. Range Queries in Trie-Structured Overlays. In *The Fifth IEEE International Conference on Peer-to-Peer Computing*. 2005.

[69] M. Dayde, L. Giraud, M. Hernandez, J.-Y. L'Excellent, M. Pantel, and C. Puglisi. An Overview of the Grid-TLSE Project. In *Proceedings of 6th International Meeting VECPAR'04, Valencia, Spain*, pages pp 851–856. June 2004.

[70] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10. USENIX Association, Berkeley, CA, USA, 2004. CR-ENS-GRID.

[71] B. Del-Fabbro, D. Laiymani, J.-M. Nicod, and L. Philippe. Data Management in Grid Applications Providers. In *Procs of the 1st Int. Conf. on Distributed Frameworks for Multimedia Applications, DFMA'2005*, pages 315–322. Besançon, France, February 2005.

[72] A. Denis, C. Perez, and T. Priol. Towards high performance CORBA and MPI middlewares for grid computing. In C. A. Lee, editor, *Proc. of the 2nd International Workshop on Grid Computing*, number 2242 in LNCS, pages 14–25. Springer-Verlag, Denver, Colorado, USA, November 2001.

[73] B. Depardon. *Automatic Deployment for Application Service Provider Environments*. Thesis, École Normale Supérieure de Lyon, October6 2010.

[74] F. Desprez. *Contribution à l'algorithme parallèle. Calcul numérique : des bibliothèques aux environnements de metacomputing*. HDR (Habilitation à Diriger les Recherches), Université Claude Bernard de Lyon. LIP. ENS-Lyon., July 2001.

[75] F. Desprez, E. Caron, and G. Le Mahec. DAGDA: Data Arrangement for the Grid and Distributed Applications. In *AHEMA 2008. International Workshop on Advances in High-Performance E-Science Middleware and Applications. In conjunction with eScience 2008*, pages 680–687. Indianapolis, Indiana, USA, December 2008.

[76] F. Desprez, J. Dongarra, A. Petitet, C. Randriamaro, and Y. Robert. Scheduling Block-Cyclic Array Redistribution. In E. D'Hollander, G. Joubert, F. Peters, and U. Trottenberg, editors, *Parallel Computing: Fundamentals, Applications and New Directions*, pages 227–234. North Holland, 1998.

[77] S. Dolev. *Self-Stabilization*. The MIT Press, 2000.

[78] M. Q. F. Desprez and F. Suter. Dynamic Performance Forecasting for Network Enabled Servers in a Metacomputing Environment. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001)*. CSREA Press, 25-28 June 2001.

[79] L. Ferreira, A. Thakore, M. Brown, F. Lucchese, H. RuoBo, L. Lin, P. Manesco, J. Mausolf, N. Memtaheni, K. Subbian, *et al. Grid Services Programming and Application Enablement*. IBM, 2004. ISBN: 0738498033. IBM Form Number: SG24-6100-00.

[80] A. Flissi, J. Dubus, N. Dolet, and P. Merle. Deploying on the Grid with DeployWare. In *CCGRID'08: Proceedings of the 8th IEEE/ACM International Symposium on Cluster Computing and the Grid*. Lyon, France, May 2008.

[81] A. Flissi and P. Merle. A Generic Deployment Framework for Grid Computing and Distributed Applications. In *GADA'06: Proceedings of the 2nd International OTM Symposium on Grid computing, High-PerformAnce and Distributed Applications*, volume 4279 of *Lecture Notes in Computer Science*, pages 1402–1411. Springer-Verlag, Montpellier, France, November 2006.

[82] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.

[83] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[84] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, *et al.* The open grid services architecture, version 1.0. In *Global Grid Forum, Lemont, Illinois, USA, GFD-I*, volume 30. 2005.

[85] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud Computing and Grid Computing 360-Degree Compared. In *2008 Grid Computing Environments Workshop*, pages 1–10. IEEE, November 2008. ISBN 978-1-4244-2860-1.

[86] W. Gentzsch. Sun Grid Engine: towards creating a compute power grid. pages 35–36. 2001.

[87] W. Goad and M. Kanehisa. Pattern Recognition in Nucleic Acid Sequences. I. A General Method for Finding Local Homologies and Symmetries. *Nucleic Acids Research*, 10(1):247–263, 1982.

[88] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load Balancing in Dynamic Structured P2P Systems. In *Proc. IEEE INFOCOM*. Hong Kong, 2004.

[89] A. S. Gokhale and D. C. Schmidt. Measuring and Optimizing CORBA Latency and Scalability Over High-Speed Networks. *IEEE Transactions on Computers*, 47:391–413, 1998.

[90] A. S. Gokhale and D. C. Schmidt. Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance. In *HICSS'98: Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences-Volume 7*, page 376. IEEE Computer Society, Washington, DC, USA, 1998. ISBN 0-8186-8251-5.

[91] R. Goldberg. Survey of virtual machine research. *IEEE Computer*, 7(6):34–45, 1974.

[92] P. Goldsack and P. Toft. SmartFrog: a Framework for Configuration. In *Large Scale System Configuration Workshop*. National e-Science Centre UK, 2001. Http://www.hpl.hp.com/research/smartfrog/.

[93] W. Goscinski and D. Abramson. Distributed Ant: A System to Support Application Deployment in the Grid. In *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*. Nov. 2004.

[94] A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse Cholesky factorization on a hypercube. *Parallel Computing*, 10:287–298, 1989.

[95] A. Halderen, B. Overeinder, and P. Sloot. Hierarchical Resource Management in the Polder Metacomputing Initiative. *Parallel Computing*, 24:1807–1825, 1998.

[96] R. S. Hall, D. Heimbigner, and A. L. Wolf. A cooperative approach to support software deployment using the software dock. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 174–183. IEEE Computer Society Press, Los Alamitos, CA, USA, 1999. ISBN 1-58113-074-0.

[97] T. Herault, P. Lemarinier, O. Peres, L. Pilard, and J. Beauquier. A Model for Large Scale Self-Stabilization. In IEEE, editor, *21th International Parallel and Distributed Processing Symposium, IPDPS 2007*. 2007.

[98] T. Herman and T. Masuzawa. A Stabilizing Search Tree with Availability Properties. In IEEE, editor, *Proceedings of the 5th International Symposium on Autonomous Decentralized Systems (ISADS'01)*, pages 398–405. 2001.

[99] T. Herman and T. Masuzawa. Available Stabilizing Heaps. *Information Processing Letters*, 77:115–121, 2001.

[100] J. Holland. *Adaptation in Natural and Artificial Systems*. MIT press Cambridge, MA, 1992.

[101] Z. Hou, J. Tie, X. Zhou, I. Foster, and M. Wilde. ADEM: Automating Deployment and Management of Application Software on the Open Science Grid. In IEEE, editor, *Grid 2009, 10th IEEE/ACM International Conference on Grid Computing*, pages 130–137. IEEE, Banff, October 13-15 2009.

[102] P. Inc. Administering Platform LSF. *Version*, 6:451–460, 2005.

[103] M. Jette and M. Grondona. SLURM: Simple linux utility for resource management. *Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 44–60, 2003.

[104] D. R. Karger and M. Ruhl. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. In *IPTPS*, pages 131–140. 2004.

[105] K. Keahey, M. Tsugawa, A. Matsunaga, and J. A. Fortes. Sky Computing. *IEEE Internet Computing Journal*, 13(5):43–51, Sept 2009.

[106] T. Kichkaylo, A. Ivan, and V. Karamcheti. Constrained component deployment in wide area networks using AI planning techniques. In *International Parallel and Distributed Processing Symposium*. Apr. 2003.

[107] T. Kichkaylo, A. Ivan, and V. Karamcheti. Sekitei: An AI Planner for Constrained Component Deployment in Wide-Area Networks. Technical Report TR2004-851, Department of Computer Science Courant Institute of Mathematical Sciences, New York University, March 2004.

[108] T. Kichkaylo and V. Karamcheti. Optimal resource aware deployment planning for component based distributed applications. In *The 13th High Performance Distributed Computing*. June 2004.

[109] J. Kim and S. Kim. A multifrontal solver combined graph partitioners. *AIAA Journal*, 38(8):964–970, 1999.

[110] S. J. Kim, C. S. Lee, J. H. Kim, M. Joh, and S. Lee. IPSAP: A High-performance Parallel Finite Element Code for Large-scale Structural Analysis Based on Domain-wise Multifrontal Technique. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 32. IEEE Computer Society, Washington, DC, USA, 2003. ISBN 1-58113-695-1.

[111] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230. 2007.

[112] C. Knight, S. Knight, N. Massey, T. Aina, C. Christensen, D. Frame, J. Kettleborough, A. Martin, S. Pascoe, B. Sanderson, D. Stainforth, and M. Allen. Association of parameter, software and hardware variation with large scale behavior across 57,000 climate models. *Proceedings of the National Academy of Sciences*, 104:12259–12264, 2007.

[113] I. Korf. *BLAST*. O'Reilly, 2003.

[114] T. Kosar and M. Livny. Stork: Making Data Placement a First Class Citizen in the Grid. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings*, pages 342–349. 2004.

[115] N. Krasnogor and J. Smith. A Java Memetic Algorithms Framework. In *Workshop Program, Proceedings of the 2000 Genetic and Evolutionary Computation Conference. Morgan Kaufmann*. 2000.

[116] S. Lacour. *Contribution à l'Automatisation du Déploiement d'Applications sur des Grilles de Calcul*. Thèse de doctorat, Université de Rennes 1, IRISA, Rennes, France, December 2005.

[117] S. Lacour, C. Pérez, and T. Priol. Deploying CORBA Components on a Computational Grid: General Principles and Early Experiments Using the Globus Toolkit. In *2nd International Working Conference on Component Deployment*. May 2004.

[118] S. Lacour, C. Pérez, and T. Priol. Generic Application Description Model: Toward Automatic Deployment of Applications on Computational Grids. In *GRID'05: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*. Springer-Verlag, Seattle, WA, USA, November 2005.

[119] E. Laure and B. Jones. Enabling grids for e-science: The egee project. *Grid Computing: Infrastructure, Service, and Applications*, page 55, 2009.

[120] J. Ledlie and M. I. Seltzer. Distributed, Secure Load Balancing with Skew, Heterogeneity and Churn. In *INFOCOM*, pages 1419–1430. 2005.

[121] M. Cai and M. Frank and J. Chen and P. Szekely. MAAN: A Multi-Attribute Addressable Network for Grid Information Services. 2(1):3–14, March 2004.

[122] C. Martin and O. Richard. Parallel Launcher for Cluster of PC. In *Parallel Computing, Proceedings of the International Conference*. Sep. 2001.

[123] M. Massie, B. Chun, and D. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, July 2004.

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

[125] P. Maymounkov and D. Mazieres. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Proceedings of IPTPS02*. Cambridge, USA, March 2002.

[126] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998. ISBN 0262631857.

[127] J. Montagnat, B. Isnard, T. Glatard, K. Maheshwari, and M. Blay-Fornarino. A data-driven workflow language for grids based on array programming principles. In *Workshop on Workflows in Support of Large-Scale Science(WORKS'09)*, , pages 1–10. November 2009.

[128] R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente. Elastic management of cluster-based services in the cloud. In *ACDC '09: Proceedings of the 1st workshop on Automated control for datacenters and clouds*, pages 19–24. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-585-7.

[129] C. Morin, R. Lottiaux, G. Vallée, P. Gallard, D. Margery, J. Berthou, and I. Scherson. Kerrighed and data parallelism: Cluster computing on single system image operating systems. In *cluster*, pages 277–286. IEEE, 2004.

[130] G. Mounie, C. Rapine, and D. Trystram. Efficient approximation algorithms for scheduling malleable tasks. In *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, pages 23–32. ACM, 1999.

[131] 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.

[132] S. Needleman and C. Wunsch. A General Method Applicable to the Search for Similarities in the Amino Acid Sequences of Two Proteins. *Journal of Molecular Biology*, pages 443–453, 1970.

[133] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. Eucalyptus: A Technical Report on an Elastic Utility Computing Architecture Linking Your Programs to Userful Systems. (2008-10), 2008.

[134] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus Open-source Cloud-computing System. In *Cloud Computing and Its Applications (CCA-08)*. Chicago, October 22-23 2008.

[135] S. Oaks, B. Traversat, and L. Gong. *JXTA in a Nutshell*. O'Reilly & Associates, Inc., 2002. ISBN 0-596-00236-X.

[136] 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.

[137] OW2 Consortium. The Fractal Project, 2010. `http://fractal.ow2.org`.

[138] P. Petr Tůma. CCPsuite, 2010. `http://d3s.mff.cuni.cz/~ceres/prj/CCPsuite`.

[139] A. Prenneis Jr. Loadleveler: Workload management for parallel and distributed computing environments. *Proceedings of Supercomputing Europe (SUPEUR)*, 176, 1996.

[140] 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.

[141] 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.

[142] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Prefix Hash Tree: an Indexing Data Structure over Distributed Hash Tables. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*. 2004.

[143] R. Raman, M. Solomon, M. Livny, and A. Roy. *The ClassAds Language*, chapter Scheduling in the Grid Application Development Software Project, pages 255–270. Kluwer Academic Publisher, September 2003.

[144] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Adressable Network. In *ACM SIGCOMM*, pages 161–172. 2001.

[145] D. Raz and Y. Shavitt. New Models and Algorithms for Programmable Networks. *Computer Networks*, 38(3):311–326, 2002.

[146] D. Reed and C. Mendes. Intelligent Monitoring for Adaptation in Grid Applications. In *Proceedings of the IEEE*, volume 93, pages 426–435. February 2005.

[147] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-To-Peer Systems. In *International Conference on Distributed Systems Platforms (Middleware)*. November 2001.

[148] F. Sabatier, A. De Vivo, and S. Vialle. Grid Programming for Distributed Remote Robot Control. In *WETICE*, pages 358–363. IEEE Computer Society, 2004. ISBN 0-7695-2183-5.

[149] J. Santoso, G. van Albada, B. Nazief, and P. Sloot. Simulation of Hierarchical Job Management for Meta-Computing Systems. *Int. Journal of Foundations of Computer Science*, 12(5):629–643, 2001.

[150] C. Schmidt and M. Parashar. Enabling Flexible Queries with Guarantees in P2PSystems. *IEEE Internet Computing*, 8(3):19–26, 2004.

[151] A. Segall. Distributed Network Protocols. *IEEE Transactions on Information Theory*, IT-29:23–35, 1983.

[152] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In M. Parashar, editor, *Grid Computing - GRID 2002, Third International Workshop*, volume 2536 of *LNCS*, pages 274–278. Springer, Baltimore, MD, USA,, November 2002.

[153] B. C. Shu, Y. Ooi, K. Tan, and A. Zhou. Supporting Multi-Dimensional Range Queries in Peer-to-Peer Systems. In *Peer-to-Peer Computing*, pages 173–180. 2005.

[154] G. Singh, E. Deelman, G. Mehta, K. Vahi, M.-H. 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: Proc. of the 2005 ACM symposium on Applied computing*, pages 680–686. ACM Press, New York, NY, USA, 2005. ISBN 1-58113-964-0.

[155] T. Smith and M. Waterman. Identification of Common Molecular Subsequences. *J. Mol. Bwl*, 147:195–197, 1981.

[156] D. P. Solutions. Simplifying System Deployment using the Dell OpenManage Deployment Toolkit, October 2004.

[157] B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster. Virtual Infrastructure Management in Private and Hybrid Clouds. *IEEE Internet Computing*, 13(5):14–22, September 2009.

[158] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup service for Internet Applications. In *ACM SIGCOMM*, pages 149–160. 2001.

[159] Y. Tanimura, K. Seymour, E. Caron, A. Amar, H. Nakada, Y. Tanaka, and F. Desprez. *Interoperability Testing for The GridRPC API Specification*. Open Grid Forum, May 2007. OGF Reference: GFD.102.

[160] C. Team. The directed acyclic graph manager. `http://www.cs.wisc.edu/condor/dagman`.

[161] G. Team. DIET User's Manual. http://graal.ens-lyon.fr/DIET.

[162] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[163] The JXTA project. http://www.jxta.org.

[164] H. Topcuouglu, S. Hariri, and M. you Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274, 2002.

[165] B. Traversat, M. Abdelaziz, and E. Pouyoul. A Loosely-Consistent DHT Rendezvous Walker. Technical report, Sun Microsystems, Inc, March 2003.

[166] S. Vadhiyar and J. Dongarra. Self Adaptability in Grid Computing. *Concurrency and Computation: Practice and Experience*, 17(2-4), 2005.

[167] R. Van Engelen and K. Gallivan. The gSOAP toolkit for web services and peer-to-peer computing networks. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 128. IEEE Computer Society Washington, DC, USA, 2002.

[168] L. Vaquero, L. Rodero-Merino, J. Caceres, and M. Linder. A Break in the Clouds: Towards a Cloud Definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2009.

[169] C. A. Varela, P. Ciancarini, and K. Taura. Worldwide computing: Adaptive middleware and programming technology for dynamic Grid environments. *Scientific Programming Journal*, 13(4):255–263, December 2005. Guest Editorial.

[170] Y. Wang, A. Carzaniga, and A. L. Wolf. Four Enhancements to Automated Distributed System Experimentation Methods. In *ICSE'08: Proceedings of the 30th international conference on Software engineering*, pages 491–500. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-079-1.

[171] Y. Wang, M. J. Rutherford, A. Carzaniga, and A. L. Wolf. Automating Experimentation on Distributed Testbeds. In *ASE 2005: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. Long Beach, CA, November 7-11 2005.

[172] B. Ward. *The book of VMware: the complete guide to VMware workstation.* No Starch Press, pub-NO-STARCH:adr, 2002. ISBN 1-886411-72-7.

[173] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *The Journal of Future Generation Computing Systems*, 15(5-6):757–768, 1999.