HAL
open science

GoDIET: a tool for managing distributed hierarchies of DIET agents and servers

Eddy Caron, Holly Dail

To cite this version:

Eddy Caron, Holly Dail. GoDIET: a tool for managing distributed hierarchies of DIET agents and servers. [Research Report] RR-5520, LIP RR-2005-06, INRIA, LIP. 2005, pp.19. inria-00070486

HAL Id: inria-00070486

https://inria.hal.science/inria-00070486v1

Submitted on 19 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

GoDIET: a tool for managing distributed hierarchies of DIET agents and servers

Eddy Caron — Holly Dail

N° 5520

March 2005

Thème NUM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--5520--FR+ENG

GoDIET: a tool for managing distributed hierarchies of DIET agents and servers

Eddy Caron , Holly Dail

Thème NUM — Systèmes numériques
Projet GRAAL

Rapport de recherche n° 5520 — March 2005 — 19 pages

Abstract: The Distributed Interactive Engineering Toolbox (DIET) is an Application Service Provider (ASP) platform providing remote execution of computational problems on distributed, heterogeneous resources. Traditional ASP toolkits are based on a single, centralized scheduling agent coordinating computation requests from clients with service offerings from servers. DIET is based on a hierarchy of agents that collaborate to perform scheduling decisions; we are exploring the benefits of hierarchical agent architectures for scalability and adaptation to heterogeneous network performance. This paper describes GoDIET, a new tool for the configuration, launch, and management of DIET on computational grids. Users of GoDIET write an XML file describing their available compute and storage resources and the desired overlay of DIET agents and servers onto those resources. GoDIET automatically generates and stages all necessary configuration files, launches agents and servers in appropriate hierarchical order, reports feedback on the status of running components, and allows shutdown of all launched software. We present a series of experiments that permit the evaluation of the performance of GoDIET for several launch and management approaches. We also evaluate the robustness of the DIET platform for a large number of servers.

Key-words: Deployment, PSE, Network enabled server, Grid computing

This text is also available as a research report of the Laboratoire de l'Informatique du Parallélisme `http://www.ens-lyon.fr/LIP`.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

GoDIET : un outil pour la gestion de hiérachies distribuées d'agents et de serveurs DIET

Résumé : Dans cet article, nous exposons les travaux menés autour de la configuration, du lancement et de la gestion de DIET, un intergiciel de type ASP (*Application Service Provider*) pour la grille. La difficulté de cette tâche repose sur l'architecture de DIET qui est distribuée et hiérarchique. Le besoin de disposer de ce type d'outil est renforcé par la diversité et le nombre des éléments de cet intergiciel. Dans cet article, nous présenterons GoDIET, un nouvel outil adapté aux contraintes de DIET, cependant les concepts mis en œuvre restent valides pour tout environnement distribué et hiérarchique. Le principe de fonctionnement de GoDIET sera détaillé au travers de son utilisation pour la gestion de DIET et du LogService, un service externe pour la gestion de traces d'éléments distribués. Enfin, nous présentons une série d'expérimentations qui permettent d'évaluer la performance et l'efficacité de GoDIET. Dans cette même série d'expériences la robustesse de la plate-forme DIET sur un grand nombre de serveurs sera également étudiée.

Mots-clés : Déploiement, PSE, Calcul sur la grille

1 Introduction

The Distributed Interactive Engineering Toolbox (DIET) is an Application Service Provider (ASP) platform providing remote execution of computational problems on distributed, heterogeneous resources. ASP toolkits are typically based on three components: **servers** that are attached to compute resources and provide solutions to computational problems, **clients** that submit problems they need solved, and a centralized **agent** that coordinates computation requests with server availability. Instead of using a single agent, DIET is based on a hierarchy of agents that collaborate to perform scheduling decisions; we are exploring the benefits of hierarchical agent architectures for scalability and adaptation to heterogeneous network performance.

In this paper we are interested in automated approaches for launching and managing DIET agent hierarchies across computational grids. The launch of DIET on computational grids presents a number of problems shared with many other grid solutions: how to quickly and reliably launch a large number of components, how to interface with a variety of resource environments (ranging from direct execution with ssh to launching via a batch system), how to regain control of launched processes to shut them down, and how to identify and react to failures in the launch process. The launch of DIET is further complicated by the fact that agents and servers must coordinate with the rest of the deployment; specifically, the chosen hierarchy is encapsulated in agent and server configuration files and the launcher must ensure that parent agents are fully functional before launching agents and servers that are further down in the hierarchy. We will use the word **deployment** to encapsulate all of the above general and DIET-specific launch and management issues; although deployment is an overloaded term, and thus not a desirable choice, there are scant alternatives for encapsulating these issues in one word.

Traditionally, users of DIET have either launched agents and servers by hand, or written scripts to manage the launch. These approaches place serious limitations on the diversity, scale, and number of experiments that can be feasibly performed. Since DIET itself is currently stable and provides good performance at small scale, support for larger scale experiments is vital for further testing.

This paper describes GoDIET, a new tool we have developed for the configuration, launch, and management of DIET on computational grids. GoDIET is written in Java and users write an XML file describing their available compute and storage resources and the desired overlay of agents and servers onto those resources. GoDIET automatically generates and stages all necessary configuration files, launches agents and servers in appropriate hierarchical order, reports feedback on the status of running components, and allows shutdown of all launched software.

This paper is organized as follows. In Section 2 we discuss related tools. In Section 3 we provide a more complete overview of DIET, as well as services for logging and visualization we use in conjunction with DIET. In Section 4 we describe in detail the deployment approach used by GoDIET. Then, in Section 5 we present a series of experiments that compare the performance of GoDIET with and without feedback on the status of launched components.

We also evaluate the robustness of DIET platforms that incorporate a large number of servers, thus demonstrating also that GoDIET supports experimentation at a large scale.

2 Related work

A variety of toolkits are freely available that can be used for launching programs on computational grids; we briefly discuss some of these toolkits below. We wanted GoDIET to satisfy some very specific DIET needs such as respecting a DIET hierarchy in the launch order, using LogService feedback to manage the launch, and in automatic generation of configuration files that encapsulate each elements's role in the hierarchy. We did not find any currently existing toolkits that manage these tasks in a generalized way. Thus, rather than make very heavy modifications of an existing toolkit, we decided to purpose-build a deployment tool but we based our approach on existing tools wherever possible.

JXTA Distributed Framework (JDF) [1] is designed to facilitate controlled testing of JXTA-based peer-to-peer systems under a variety of scenarios including emulated failures. JDF uses an XML file for descriptions of the resources and jobs to be run, and deployment is based on a regular Java Virtual Machine, a Bourne shell, and `ssh` or `rsh`. JDF is focused on launching Java-based programs and relies on several Java features during the launch, thus it is not directly applicable to launching DIET. However, we are currently collaborating in an effort to integrate JuxMem [8], a JXTA-based distributed memory project, in DIET. We plan to use GoDIET to coordinate the launch of DIET and JuxMem together, but GoDIET will call existing JDF interfaces to manage the JuxMem deployment. JDF also includes a notion of profiles that could be used to simplify the GoDIET XML by describing groups of agents or servers as a collection.

APST (AppLeS Parameter Sweep Template) [5] provides large-scale deployment of applications based on the parameter sweep model. Resources, jobs, and parameters are described in an XML file and APST manages the staging of files, scheduling of jobs, and retrieval of results. APST is a very general solution that can be used for a wide variety of purposes; APST is not however designed for middleware deployments structured with automatically-generated configuration files. Elagi [7] is a library that provides compiled programs with easy access to grid services; essentially, Elagi packages those parts of the APST code-base that can be easily re-used by other projects. We plan to rewrite those portions of GoDIET that use `ssh` and `scp` to instead use Elagi's generic process spawning and batch submission interfaces; this will provide GoDIET the ability to deploy DIET via `ssh`, `globusrun`, or a large number of batch schedulers including LSF, PBS, and Condor.

ADAGE (Automatic Deployment of Applications in a Grid Environment) [9] is designed to automatically determine an appropriate application deployment and then enacts the launch. Applications can be either a CORBA component assembly or an MPICH-G2 parallel application and the Globus Toolkit Version 2 is used as the grid access middleware. ADAGE is a promising approach, but for our purposes there are several limitations: we want to deploy DIET on a large range of systems where Globus may not be installed or the user may not have the appropriate Globus-specific security certificates, ADAGE does

not provide the application-specific configuration file generation need for DIET, and the ADAGE software is not yet available. However, we plan to follow the development of this deployment approach to see where future integration and collaboration may be possible.

3 DIET & Related services

DIET [3] is a toolkit for building applications based on remote computational servers. The goal of DIET is to provide a sufficiently simple interface to mask the distributed infrastructure while providing users with access to greater computational power. To add functionality to DIET or simplify its usage, DIET is often used in conjunction with related services for object location, distributed trace collection, or platform visualization. We summarize DIET in the next subsection and these related services in Section 3.2.

3.1 DIET architecture

DIET can use a hierarchy of agents to provide greater scalability for scheduling client requests on available servers. The collection of agents uses a broadcast / gather operation to find and select amongst available servers while taking into account locations of any existing data (which could be pre-staged due to previous executions), loads on available servers, and application-specific performance predictions.

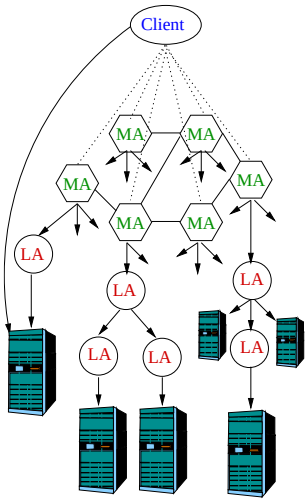

Figure 1: General view of DIET architecture.

Figure 1 provides an overview of the DIET architecture. The **Master Agent** (MA) must be launched before all other agents and servers and serves as the main *portal* to the DIET

hierarchy; all agents and servers are thereafter attached to the MA via a tree at launch-time, though the type of tree is not constrained. All other agents are called **Local Agents** (LA) and can be included in the hierarchy for scalability or to provide a better mapping to the underlying network architecture. **Servers** are attached to each computational resource and either perform the actual computation directly or launch another binary to perform the computation (e.g. in the case of a server on a front-end node of a batch system). Servers also aid in the scheduling process by providing performance predictions to agents during the server selection process. While the top of the tree must be an MA, any number of LAs can be connected in an arbitrary tree and servers can be attached to the MA or to any of the LAs.

The **client** is the application interface by which users can submit problems to DIET. Many clients can connect to DIET at once and each can request the same or different types of services by specifying the problem to be solved. Clients can use either the synchronous interface where the client must wait for the response, or the asynchronous interface where the client can continue with other work and be notified when the problem has been solved.

Problem solution proceeds as follows. The DIET client submits the problem to the MA. Agents maintain a simple list of subtrees containing a particular service; thus the MA broadcasts the request on all subtrees containing that service. Other agents in the hierarchy perform the same operation, forwarding the request on all eligible subtrees. When the request reaches the server-level, each server calls their local FAST [6] service to predict the execution time of the request on this server. These predictions are then passed back up the tree where each level in the hierarchy sorts the responses to minimize execution time and passes a subset of responses to the next higher level. Finally the MA returns one or several proposed servers to the client. The client then submits the problem directly to the chosen server.

Note that this description of the agent/server architecture focuses on the common-case usage of DIET. An extension of this architecture is also available [4] that uses JXTA peer-to-peer technology to allow the forwarding of client requests between MAs, and thus the sharing of work between otherwise independent DIET hierarchies. We are investigating the possible benefits of this peer-to-peer extension of DIET for scalability, fault-tolerance, and collaborations between administrative domains that desire greater independence. Since this architecture is still experimental, we did not consider it for the design of GoDIET.

3.2 Associated services

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

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

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

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

Figure 2 provides an overview of the interactions between a running DIET platform, LogService, and VizDIET. In the next section we describe the third external service in the figure - GoDIET.

4 GoDIET

The goal of GoDIET is to automate the deployment of DIET platforms and associated services for diverse grid environments. Specifically, GoDIET automatically generates configuration files for each DIET component taking into account user configuration preferences and the hierarchy defined by the user, launches complimentary services (such as a name service and logging services), provides an ordered launch of components based on dependencies defined by the hierarchy, and provides remote cleanup of launched processes when the deployed platform is to be destroyed.

Key goals of GoDIET included portability, the ability to integrate GoDIET in a graphically-based user tool for DIET management, and the ability to communicate in CORBA with LogService; we have chosen Java for the 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 like LogService, the physical machines to be used, the disk space available on these machines, and the

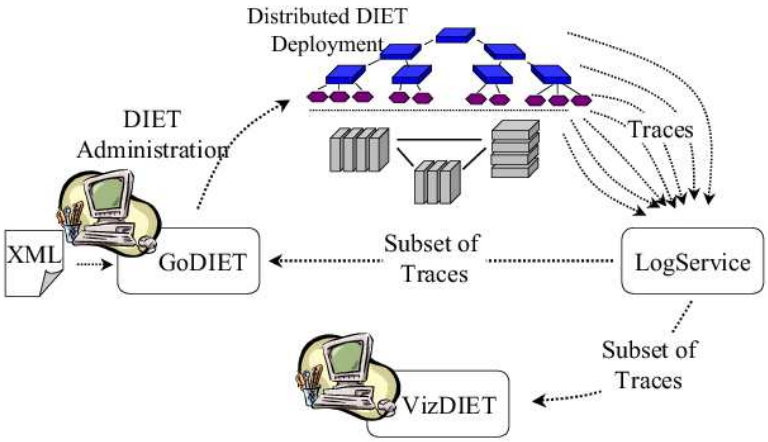

Figure 2: Interaction of GoDIET, LogService, and VizDIET to assist users in controlling and understanding DIET platforms.

configuration of paths for the location of needed binaries and dynamically loadable libraries. The file format provides a strict separation of the resource description and the deployment configuration description; the resource description portion must be written once for each new grid environment, but can then be re-used for a variety of deployment configurations.

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

We use `scp` and `ssh` to provide secure file transfer and task execution. `ssh` is a tool for remote machine access that has become 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. To illustrate the approach used, an example of the type of command used by GoDIET follows.

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

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

A example input XML file for GoDIET is provided in Appendix A with commentary explaining the format of each component. This example should prove a useful reference for the following discussions.

4.1 DIET Deployment

The DIET platform is constructed following the hierarchy of agents and SeDs; Figure 3 provides an overview of the DIET startup process. The very first element to be launched during deployment is the naming service; all other elements are provided the hostname and port at which the naming service can be found. Afterwards, deployed elements can locate other elements on the grid using only the element's string-based name and the contact information for the naming service. After the naming service, the MA is launched; the MA is the root of the DIET hierarchy and thus does not register with any other elements. After the MA, all other DIET elements understand their place in the hierarchy from their configuration file which contains the name of the element's parent. Users of GoDIET specify the desired hierarchy in a more intuitive way via the naturally hierarchical XML input file to GoDIET (see the `<diet_hierarchy>` section of Appendix A). Based on the user-specified hierarchy, GoDIET automatically generates the appropriate configuration files for each element, and launches elements in a top-down fashion to respect the hierarchical organization.

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

DIET provides the features and flexibility to allow a wide variety of deployment configurations, even in difficult network and firewall environments. For example, for platforms without DNS-based name resolution or for machines with both private and public network

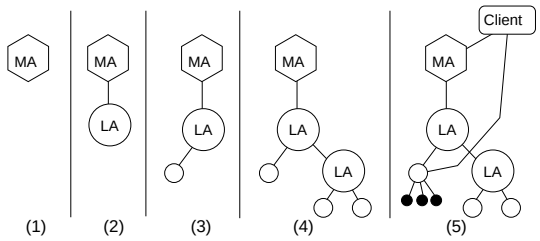

Figure 3: Launch of a DIET platform.

interfaces, elements can be manually assigned an *endpoint* hostname or IP in their configuration files; when the element registers with the naming service, it specifically requests this endpoint be given as the contact address during name lookups. Similarly, an *endpoint* port can be defined to provide for situations with limited open ports in firewalls. These specialized options are provided to DIET elements at launch time via their configuration files; GoDIET supports these configuration options via more user-intuitive options in the input XML file and then automatically incorporates the appropriate options while generating each element's configuration file. For large deployments, it is key to have a tool like GoDIET to make practical use of these features.

4.2 LogService deployment and feedback-based launch

There are two interesting aspects to the deployment of LogService: the configuration and launch of the service itself, and an interesting usage of the service by GoDIET after its launch.

When the user requests inclusion of LogService in the deployment, it is launched after the naming service and before any DIET elements. GoDIET incorporates in each element's configuration file that log messages are to be sent to the LogService; thus, when elements startup they register their existence and their hierarchical position immediately with the LogService. The user may also specify in the GoDIET XML file that they wish to run DIETLogTool as well. This tool registers with the LogService as a listener and exports log messages to a text-based file. This text-based file provides a handy record of the deployment and the performance of requests processed by the deployment; it can also be used for post-mortem analysis by VizDIET.

A sticky point for the proper deployment of DIET is that each element is deployed by a very fast ssh call that returns without knowledge of the start-up state of the element. However, it is important to ensure that an element has registered with the naming service before launching dependent elements. For relatively uniform network environments it is sufficient for GoDIET to wait for a short time (i.e. `sleep`) before deploying each component with dependencies. For the highly dynamic network environments of some grids, a more adaptive approach is necessary.

We have thus incorporated a feedback-based launch approach that allows verification of each element's status before launching the next element. After launching LogService, GoDIET registers with the LogService as a listener for messages indicating the arrival of new DIET elements. For the launch of all other elements GoDIET simply waits for verification from the LogService that the element is running correctly. This approach is highly adaptable to heterogeneous network environments: in fast network environments the launch proceeds very quickly as elements register with the naming service and the hierarchy quickly and log verifications arrive right away, while in slower network environments, slow log verification arrival indicates that element registration with the naming service and the hierarchy is slow. This direct feedback on deployment status provides a very dependable deployment approach without the need to fine-tune waiting periods for each new deployment environment.

5 Experiments

We present a group of experiments designed to evaluate the performance and efficacity of GoDIET for the deployment of DIET and associated services at a large scale. These experiments also allow us to evaluate the robustness of the DIET platform itself.

Our experimental approach is as follows: select a testbed and a DIET hierarchy to test, describe the platform in a XML file, measure the time taken by GoDIET to launch the platform, launch a series of tests to evaluate the efficicacy of the resulting DIET platform, and then use GoDIET to cleanly shut-down the platform.

A key aspect of this process is the coordination by GoDIET of the launch of components with dependencies. To build a properly deployed DIET platform, each agent must register in the naming service before a dependent component tries to link to it. Logically, as leaf nodes in the DIET hierarchy, the launch of DIET servers do not require any coordination beyond ensuring that the parent exists. We test three approaches to satisfying component dependencies:

Fixed wait: This is the simplest approach, and involves simply sleeping for a fixed period before launching dependent components. Our experiments are performed with a wait of 3 seconds after omniNames and LogCentral, 2 seconds after each agent, and 1 second after each server. These times were chosen after a small number of evaluation experiments, and are likely overgenerous in the environments of our experiments.

Feedback: This approach uses real-time feedback from LogService to guide the launch process. GoDIET waits for verification that a component has registered with the logging service before launching other components.

Aggressive: This approach uses the feedback approach for the launch of omniNames, LogCentral, and the DIET agents, but includes no pause between servers. This approach is based on the logic that since there are no dependencies between servers, it is reasonable to launch them in parallel (or in extremely fast succession).

Cluster	Site	Nodes	# CPUs	Proc Type	Mem
PARACI	Rennes	64	4	Xeon 2.4 GHz	1 GB
LS	Lyon	16	2	Pentium III 1.4 GHz	1 GB
Cristal	Rocquencourt	13	2	Xeon 2.8 GHz	2 GB
Cobalt	Grenoble	9	2	Pentium III 1.4 GHz	1 GB

Table 1: Description of the experiment platform. The number of processors and the memory size are given per node.

Once a platform has been launched by GoDIET, we want to verify several aspects of the resulting DIET platform: does the agent hierarchy perform the scheduling process efficiently and are all of the servers functional and responsive to client requests. To test these aspects of DIET, we used a specialized DIET server that ensures the DIET agent hierarchy enforces round-robin behavior among servers when scheduling. While servers normally return predicted execution time, this specialized server returns time since last job executed. Round-robin behavior allows us to verify the functionality of all servers in a platform in a short time. For the evaluation of the DIET platform, we are also interested in evaluating the overheads of using the DIET platform for scheduling and job execution. Thus, we chose a very simple problem for our client / server pair that involves a very small data transfer and a small amount of computation. To evaluate DIET overhead, we take an upper bound by assuming that all time taken by this client / server pair is DIET overhead.

For a platform with S servers, we run the client program with S sequential repetitions. DIET clients use a 2-phase process to interact with DIET: they request a suitable server from the hierarchy (the scheduling phase), and then they submit the job directly to the server (the computation phase). We consider that if we have S valid scheduler responses from the agent hierarchy, the agent hierarchy is running well. We then verify that each of the S requests was assigned a unique server (as required by our choice of round-robin). If so, and if the computation phase of all S requests completed successfully, we consider all S servers to be functioning correctly.

Server malfunction is normally difficult to observe directly with DIET since DIET is tolerant of failing servers and the scheduling system simply directs clients to functioning servers. However, by checking that a unique server was used for each of S requests on S servers, we can see that if any server was chosen more than once, another server must have failed. For each platform studied we measure the total time for the execution of all S requests and divide by the number of requests to find the average latency of DIET scheduling + computation.

For these experiments we use a grid with 4 sites, 102 machines, and 342 processors in total. Table 1 provides details on this platform. This large-scale grid is based on VTHD (Vraiment Très Haut Debit), a high-bandwidth network in France.

5.1 Evaluation of launch performance

For these experiments, the type of DIET hierarchy is fixed and we vary the number of servers. The hierarchy is composed of an MA on the cluster `ls` and an LA on each of the other four sites. Depending on the experiment, we then add between 2 and 32 servers at each site (thus between 8 and 128 servers in total). Figure 4 presents the time required to launch these different platforms. Each platform was tested three times and values plotted are the mean and standard deviation of these three runs.

The time for launch is strongly dependent on the number of servers included in the deployment. The cost of the *fixed wait* approach is clearly higher than the other approaches, while the *aggressive* approach is the fastest since there is no waiting for the launch of the servers. The *feedback* approach has moderate performance and the performance variation is higher because the approach is so strongly linked to feedback data.

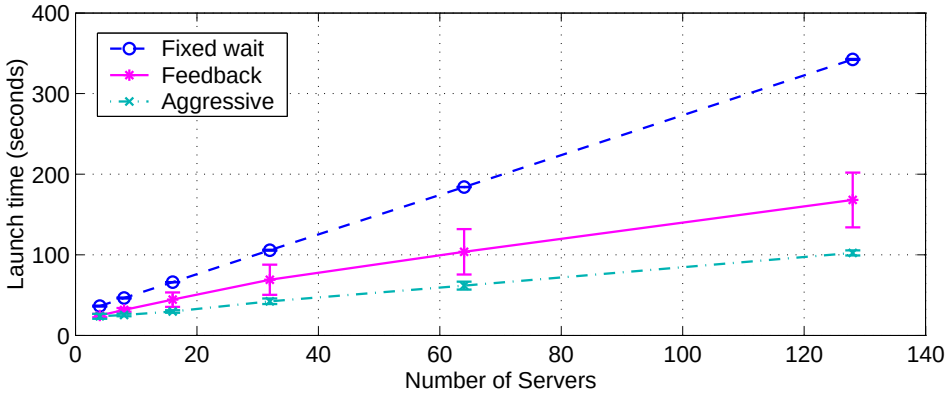

Figure 4: The time for platform launch as a function of the number of servers desired.

5.2 Evaluation of DIET performance and stability

Table 2 summarizes the results of our experiments. The average DIET request overhead is similar across the three launch approaches, and increases with the number of servers. The second type of data indicates the percentage of client requests that were not succesfully served. Any value under 100% indicates either a failure of the agent scheduling hierarchy to find a satisfactory server or a failure of the solve request after it was submitted to the chosen server. We only see this type of failure with the *aggressive* launch approach, and then the failure is relatively catastrophic. We theorize that there is a problem with the ability of DIET agents to manage a large number of server registrations at once and that in the case of problems during registration, the agent-level data structures concerning the servers can be irrecoverably damaged. We plan to investigate this problem in detail as parallel launch

	4 SeDs	8 SeDs	16 SeDs	32 SeDs	64 SeDs	128 SeDs
GoDIET	Mean latency for DIET requests (sec)					
Fixed wait	0.50	0.63	0.69	1.04	1.59	2.73
Feedback	0.50	0.63	0.71	0.96	1.37	2.54
Aggressive	0.33	0.57	0.63	1.09	1.43	2.08
	Percentage of failed requests					
Fixed wait	0%	0%	0%	0%	0%	0%
Feedback	0%	0%	0%	0%	0%	0%
Aggressive	0%	29.2%	0%	32.3%	33.3%	0%
	Percentage of servers inaccessible to client					
Fixed wait	0%	0%	0%	0%	0%	0%
Feedback	0%	0%	2.1%	3.1%	0%	0%
Aggressive	8.3%	29.2%	4.2%	33.3%	36.5%	3.4%

Table 2: Results on DIET performance and stability.

of servers remains a very rapid deployment approach. For the meantime, we conclude that the *Aggressive* approach is not a good choice. The last metric, the percentage of servers inaccessible to clients, is a direct measure of how many servers failed. We calculate this statistic by evaluating to what extent the round-robin scheduling algorithm was able to use every server in its turn.

To conclude, while the *aggressive* launch approach is the fastest, it does not result in stable hierarchies. The *feedback* approach is both relatively fast, stable, and directly adaptable to diverse network environments.

6 Conclusion

In this article we have presented GoDIET, a tool for the hierarchical deployment of distributed DIET agents and servers. We described DIET and the services that can be added to DIET to augment its functionality. We discussed in detail the launch process for DIET and its services, and how GoDIET simplifies that process. We presented experiments testing the performance of three approaches to handling inter-element dependencies in the launch process: usage of feedback from logservice to guide the launch, fixed sleep period between dependent elements, and an aggressive approach that uses feedback for all agents but launches all servers without waiting. Based on experimental results we conclude that using feedback is the most effective approach.

There are several areas we plan to pursue in future work. First, we plan to build a visual tool that will allow users to build a graphical model of their desired deployment and export this model to GoDIET for launch. We also envision enabling interactive deployment and reconfiguration whereby users can dynamically reconfigure portions of the DIET deployment. Second, we plan to integrate this manual design tool with a model that optimization

deployment plans based on resource and application characteristics. An initial model has been described in [2] and we plan to extend and adapt this model to usage in real-world deployment scenarios.

References

[1] Gabriel Antoniu, Luc Bougé, and Mathieu Jan. Juxmem: An adaptive supportive platform for data sharing on the grid. In *IEEE/ACM Workshop on Adaptive Grid Middleware, held in conjunction with 12th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT 2003)*, New Orleans, September 2003.

[2] Eddy Caron, Pushpinder Kaur Chouhan, and Arnaud Legrand. Automatic Deployment for Hierarchical Network Enabled Server. In *The 13th Heterogeneous Computing Workshop (HCW 2004)*, Santa Fe. New Mexico, April 2004.

[3] Eddy Caron and Frédéric Desprez. Diet, tour d'horizon. In *Ecole thématique sur la Globalisation des Ressources Informatiques et des Données : Utilisation et Services. GridUSe 2004*, Metz. France, july 2004. Supélec.

[4] Eddy Caron, Frédéric Desprez, Franck Petit, and Cédric Tedeschi. Resource Localization Using Peer-To-Peer Technology for Network Enabled Servers. Research report 2004-55, Laboratoire de l'Informatique du Parallélisme (LIP), December 2004.

[5] Henri Casanova, Graziano Obertelli, Francine Berman, and Rich Wolski. The AppLeS Parameter Sweep Template: User-level middleware for the Grid. In *Proceedings of Supercomputing*, November 2000.

[6] F. Desprez, M. Quinson, and F. Suter. Dynamic performance forecasting for network enabled servers in a metacomputing environment. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001)*, 2001.

[7] Elagi project description. http://grail.sdsc.edu/projects/elagi/.

[8] JuxMem project description. http://www.jxta.org/universities/juxmem.html.

[9] Sébastien Lacour, Christian Pérez, , and Thierry Priol. A software architecture for automatic deployment of corba components using grid technologies. In *Proceedings of the 1st Francophone Conference On Software Deployment and (Re)Configuration (DECOR 2004)*, Grenoble, France, October 2004.

A Example GoDIET XML file with comments

```
<?xml version="1.0" standalone="no"?>

<!DOCTYPE diet_configuration SYSTEM "../GoDIET.dtd">
<!-- GoDIET.dtd defines the semantics for input xml files for DIET
     . GoDIET uses a validating parser, so if your XML does not
     follow the rules specified in the dtd it will not be accepted
     by the parser. The comments below will help you write a valid
     XML for GoDIET.  All sections are required unless marked "
     Optional" -->

<!-- The diet_configuration tags surround all of the other sections
     .  diet_configuration can optionally contain:
        - 1 "goDiet" section: configure goDIET behavior
      and must contain each of the following:
        - 1 "resources" section: define storage and compute
            resources
        - 1 "diet_services" section: define omniNames, LogCentral,
            etc.
        - 1 "diet_hierarchy" section: define your agent hierarchy
            -->
<diet_configuration>

  <!-- Optional: If desired, the goDiet section can be included to
       allow configuration of goDIET behavior. -->
  <goDiet debug="1"
          saveStdOut="yes"
          saveStdErr="no"
          useUniqueDirs="yes"/>
        <!-- Optional: debug controls the verbosity of goDIET
             output from:  0 (no debugging) -> 2 (very verbose)
              Optional: saveStdOut controls whether stdout is
                  redirected to /dev/null or to a file in your
                  remote scratch space called <componentName>.out.
              Optional: saveStdErr as above with a file named <
                  componentName>.err.
              Optional: for useUniqueDirs, yes specifies that a
                  unique subdirectory will be created under the
                  scratch space on each machine for all files
                  relevant to the run.  If no, all files are written
                   in the scratch directly. -->

  <!-- Use the resources section to define what machines you want
       to use for computation and storage, how to access those
       resources, and where to find binaries on each. You must
       include:
        - 1 "scratch" section, at least 1 "storage" section, and at
            least
        - 1 "compute" or 1 "cluster" section. -->
  <resources>
```

```
<!-- Specify a local pathname GoDIET can use as scratch space (
     e.g. temp storage of config files). You must have write
     access to the directory. -->
<scratch dir="/tmp/GoDIET_scratch"/>

<!-- Define all storage space that will be needed to run jobs
     on your compute hosts. -->
<storage label="disk1">
    <scratch dir="/tmp/run_scratch"/>
    <scp server="hostX.site1.fr" login="<your␣login␣on␣this␣
         machine>"/>
      <!-- Optional: if login is not specified, the current
           login is used. -->
</storage>
<storage label="clusterX_disk">
    <scratch dir="/tmp/run_scratch"/>
    <scp server="hostX.clusterX.fr"/>
</storage>

<!-- Define all compute hosts that you want to use. -->
<!-- Use "compute" tags for individual machines, "cluster" to
     simplify description of large numbers of machines -->
<compute label="host1" disk="disk1">
    <ssh server="host1.site1.fr" login="<your␣login>"/>
      <!-- Define hostname and login to be used for ssh access
           to machine.
            Optional: if login is not specified, the local login
                 will be used. -->
    <env path="<bindir1>:<bindir2>:..."
         LD_LIBRARY_PATH="<libdir1>:<libdir2>:..."/>
         <!-- Optional: if you don't specify a path or
              LD_LIBRARY_PATH, your default paths on the system
              will be used. You can make the path shorter by
              putting links to all your binaries in one
              directory on the system. -->
    <end_point contact="192.5.59.198"/>
         <!-- Optional: for some networks, it is necessary to
              be explicit about which IP to use to communicate
              with running components. -->
</compute>
<compute label="host2" disk="disk1">
    <ssh server="host2.site1.fr"/>
    <env path="<bindir1>:<bindir2>:..."
         LD_LIBRARY_PATH="<libdir1>:<libdir2>:..."/>
</compute>

<cluster label="clusterX" disk="clusterX_disk" login="<your␣
     login>"/>
    <env path="<bindir1>:<bindir2>:..."
         LD_LIBRARY_PATH="<libdir1>:<libdir2>:..."/>

    <node label="clusterX_host1" disk="clusterX_disk">
        <ssh server="host1.clusterX.fr"/>
```

```
            <end_point contact="192.5.80.103"/>
        </node>
        <node label="clusterX_host2" disk="clusterX_disk">
            <ssh server="host2.clusterX.fr"/>
        </node>
    </cluster>

  </resources>

  <!-- Define DIET services. Must contain 1 "omni_names" section
       and can optionally include 1 "log_central" and 1 "log_tool"
       section. Do not define "log_tool" without defining "
       log_central". -->
  <diet_services>

    <omni_names contact="<ip␣or␣hostname>" port="2810">
        <!-- Optional: If contact is given, it is used in omniORB4.
             cfg to help all other corba components find omniNames.
              For example, if you have no DNS, you should put here
             the IP address. -->
        <!-- Optional: if port is undefined, port 2809 will be used
             . -->
        <config server="clusterX_host1"
                trace_level="1"
                remote_binary="omniNames"/>
            <!-- "server" must refer to the label of one of your "
                 compute" resources defined above. GoDIET will run
                 this service on that host.
                 "trace_level" is optional. Higher values provide
                      more debugging output from OmniNames.
                 "remote_binary" is the binary name to execute for
                      the service. -->
    </omni_names>

    <log_central connectDuringLaunch="no|yes">
        <!-- Optional: If connectDuringLaunch is set to no, GoDIET
             will launch LogCentral but will not try to use
             LogCentral feedback to guide the launch. -->
        <config server="clusterX_host2"
                remote_binary="LogCentral"/>
    </log_central>
    <!-- Optional: If log_tool tag is included, the log file
         generator tool will be launched after log_central. Note:
         LogCentral must be compiled with --enable-background. -->
    <log_tool>
        <config server="clusterX_host2"
                remote_binary="DIETLogTool"/>
    </log_tool>
    <!-- Optional: If diet_statistics tag is included, GoDIET will
         use set unique statistics output file environment variable
         for every launched element. DIET must be configured with --
         enable-stats. -->
    <diet_statistics/>
```

```
  </diet_services>

  <!-- Define desired DIET agent and server hierarchy.  Must
       include at least 1 "master_agent" section; all other sections
        are optional. -->
  <diet_hierarchy>

    <!-- Define config for "master_agent". -->
    <master_agent label="MyMA">
      <!-- "label" is optional.  If defined, it is used as a prefix
            for an automatically generated name.  The system will
           name this agent "MyMA_0". -->
        <!-- See config explanation under "services" section -->
        <config server="host1"
                trace_level="1"
                remote_binary="<binary name for diet agent on this
                     server>"/>
        <!-- See "master_agent" explanation above. This agent will
             be named MyLA_0 -->
        <local_agent label="MyLA">
            <config server="host2"
                    trace_level="1"
                    remote_binary="dietAgent"/>

            <SeD label="MySeD">
                <config server="clusterX_host2"
                        remote_binary="<binary name for this diet
                             SeD on this server>"/>
                <parameters string="T"/>
            </SeD>
        </local_agent>
        <SeD label="MySeD">
            <config server="clusterX_host1"
                    remote_binary="server"/>
            <parameters string="T"/>
        </SeD>
    </master_agent>
  </diet_hierarchy>

</diet_configuration>
```

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399