PARADIS: an Adaptive Middleware for Dynamic Task Allocation in a Grid

Michel Hurfin[1], Jean-Pierre Le Narzul[2], Julien Pley[3], Philippe Raïpin Parvédy[3]

[1] INRIA Rennes / IRISA – Campus de Beaulieu, 35042 Rennes cedex – France

[2]GET ENST Bretagne/ IRISA – Campus de Rennes, 35512 Cesson-Sévigné – France

[3]University of Rennes / IRISA – Campus de Beaulieu, 35042 Rennes cedex – France

{hurfin,jlenarzu,jpley,praipinp}@irisa.fr

Abstract. *The major purpose of a Grid is to federate multiple powerful resources into a single virtual entity which can be accessed transparently and efficiently by external users. As a Grid is usually an unreliable system involving heterogeneous resources located in different geographical domains, distributed and fault-tolerant resource allocation services have to be provided. In particular when a crash occurs tasks have to be reallocated quickly and automatically, in a completely transparent way from the users' point of view. This paper presents* PARADIS, *an adaptive middleware based on a set of basic agreement services that has been integrated within an experimental Grid dedicated to genomic applications. Most of these time-consuming applications are composed of a huge number of independent tasks.*

1. Introduction

A Grid is a distributed system involving heterogeneous resources located in different geographical domains that are potentially managed by different organizations (companies, laboratories, universities, ...) or individuals. The major purpose of a Grid is to federate multiple powerful distributed resources (computers but also data storage facilities) within a single virtual entity which can be accessed transparently and efficiently by external users. Most of the time, resources aggregated within such a grid are high-performance computing resources: powerful computers and clusters but also large databases and softwares whose behavior can be tuned by selecting configuration options to fit the needs of a particular Grid user. In our study, we consider a Grid composed of resources provided by various institutions. These potential contributors are identified preliminarily and correspond to well-established institutions that agree to share their resources and to trust each other. Yet each institution keeps its independence and freedom. The decision to include or to exclude some (or even all) local resources from the Grid can be taken at any time by the local administrator without any coordination with the others. Similarly, the security policy, the maintenance requirements and the rules used to manage concurrent accesses between the Grid users and the institution's members (which may use their local resources without notifying the Grid management system) are defined locally.

In this general context, we aim at developing services that will allow a Grid user to continuously take full advantage of the computing power offered by the Grid in a simple and completely transparent manner. In this new business model, the administrators of the Grid have now the responsibility of ensuring that sufficient resources are deployed to meet the varying workload demands of the users. Whatever the circumstances, a complete transparency and a quick response time are always expected by the customers. To fulfill these two requirements adaptive control mechanisms have to be proposed on one hand to cope efficiently with the dynamic changes of the computing capacity of the Grid (even if these changes are unpredictable) and on the other hand to distribute the tasks among the resources in an efficient way (dynamic load balancing). This leads us to address two major issues that both require a continuous adaptation to the changing computing environment, namely the *Resource allocation* issue and the *dependability* issue. We propose to solve both problems in an homogeneous way using a slightly modified group concept [15]. More precisely, all distant interactions between domains corresponding to distinct organizations are managed by a small group of registered processors (exactly one per domain). Each member of this group acts as a *master* for its own domain and interacts with the other members of the group to build consistent observations (1) of current workloads in each domain and (2) of the current composition of the group. In that sense, we argue that, in a distributed system prone to failures, an agreement service is a key concept to transform several local views into a single global one without opting for a centralized control approach and thus without having a single point of failures. An agreement service allows all the domains to acquire the same set of accurate data describing the current state of the Grid. Based on this unanimous observation, each domain can locally enact the right adaptation to react to the observed changes.

In addition to the dynamic evolution of the set of resources, the proposed mechanisms have to cope with unreliable estimation of the workload of each resource (even when the set of resources is stable). First, for some particular applications, the duration of a task cannot be estimated precisely. This may create a difference between the estimated workload used by the task allocation mechanism and the real workload. Second, the administrator of a domain may refuse that his resources are exclusively devoted to the Grid. Some local applications can be launched concurrently by local members of the institution without using the Grid mechanisms. In that case the workloads of the used resources increase without any control. In all the cases, adaptive mechanisms are necessary to adjust the task allocation with regards to these unforecast workload changes.

This paper focuses only on the above mentioned aspects of the design of our grid (namely transparency, resource allocation and dependability). Additional mechanisms developed to offer a secure and interactive access to the Grid (through a standard WEB site) are not detailed herein. The grid architecture and software presented in this paper have been experimented in a grid called *GénoGRID*[1] and is dedicated to genomic applications: it federates resources belonging to genomic or bioinformatics centers dispatched in the western part of France. The amount of shared data (programs, files and databases) that can be accessed and maintained through a Grid is an important factor when evaluating the interests of Grid Computing. When a Grid is dedicated to a well-identified community of users that have mutual interests and may agree on some data workflows, the volume of common information managed within the Grid is much more important.

The overall paper is organized as follows. Following this Introduction, Section 2.

[1]This Grid has been designed in the context of a project project called "ACI GénoGRID" and founded by the French Ministry of Research [12]. This project brings together researchers in biology and computer science.

outlines the relationships with some related works. Section 3. focuses on the interactions between a user and the Grid. In particular, we detail the programming rules that have to be respected by any application conceived to be executed on our experimental Grid. Section 4. discusses the multi-levels structure of the Grid which is a key characteristic of our approach. Section 5. presents the architecture of the middleware PARADIS. Some experimental results obtained during the implementation and the use of the Grid by biologists are briefly presented in Section 6.. Finally, Section 7. concludes this paper.

2. Related Work

The projects that are the most related with our work are naturally Grid computing projects and also public-resource computing projects. Both share the objective of federating multiple computing resources for use by cpu-intensive applications. Although it should be easier to address the dependability issue in Grid computing platforms (i.e. institutional projects) than in public-ressource platforms, we observe that very few projects has included sophisticated mechanisms for tolerating and masking failures. However, we provide pointers to some projects that address, in some sense, the fault-tolerance problem.

- The OurGrid project [4, 5] is a public resource project based on a peer-to-peer approach where the user of the Grid has also to act as a provider of resources. OurGrid implements a fault-tolerance mechanism based on TCP/IP timeouts. When the broker, located on a client machine, detects that the connection with a remote machine executing tasks is broken, it reallocates the tasks on another machine. Currently, the architects of OurGrid are working on a efficient, flexible and adaptable implementation of a failure detection service as well as on an easy-to-use interface to such a service. They plan to evaluate the interests of such a work for the OurGrid project.
- In the BOINC system [2], fault-tolerance is ensured by replicating tasks execution on multiple sites (redundant computing). A ”transitioner” component is in charge of generating the results; Then, as soon as a quorum of results is reached, a ”validater” component calls an application-level function to decide if the results are consistent and to select a canonical result. The major advantage of the fault-tolerant mechanisms implemented in BOINC is that it allows to protect against failures as well as against malicious participants.
- Globus [7] and UNICORE [1] are probably the most famous systems to provide a Grid infrastructure. The facilities provided by these two well-established systems address several issues not discussed in this paper. Our goal is to study the resource allocation problem in an asynchronous and unreliable setting. Until now, failures are addressed within these systems only through the definition of a centralized failure manager that is in charge of detecting and notifying the observed crashes. We claim that this approach is not the best solution because it creates a single point of failure.

3. Access to and Use of the Grid

As indicated previously our major objective is to provide a simple and transparent access to the Grid dedicated to genomic applications. As all the biologists are not expert in computer science, all the problems related to the execution of an application have to be masked. In practice, a biologist can launch his favorite applications from anywhere through one of the identified web portals [2]. Any Grid user has to be registered first to get an account: he needs to fill a form with necessary security information and receives latter a certificate that allows to authenticate him during each session setup. This registration procedure allows to manage within the Grid private directories containing personal files (applications, private files and in particular files containing the results of previous executions, ...). Thus after the login phase a user has access to his personal environment containing only familiar information related to his own activities. At this stage, the biologist has the possibility to launch one of its applications by selecting this application among a list of previously identified applications. The preliminary registration of any application that will be executed in the Grid is mandatory in our approach and has to be done once (but not necessarily by a future user). In addition to the obvious benefits in term of security, this strategy allows to gather information about the application itself (list of parameters that are sometimes optional and may have default values, list of used databases, requirements of the application in terms of operating system or memory space, estimated volume of outputs and estimated execution times obtained during tests of the application on different resources,...). At the registration time, the provided information is logged in specific repositories. In every domain, a copy of the application code and copies of the accessed data banks are created. In every portal, a simple web page is also created to simplify the submission process: a future user will just have to indicate, via this web form, the values of the input parameters and the locations of the input files in his private directory. Once the submission is done, the Grid user has no more to interact with the Grid to ensure the completion of his application. Even when the computing capacity of the Grid changes dynamically in a predictable or unpredictable manner (voluntary insertion or withdrawal of local computation resources, unfair sharing of resources between members of an institution and Grid users, crashes of some resources, ...) reconfigurations are performed automatically without the help of the Grid user. When his execution terminates the Grid user is informed by an email. Yet at any time, he has also the possibility to consult the progress of the execution on the web: the number of tasks already executed and an estimation of the number of remaining tasks are provided.

To benefit from the fact that many genomic applications can easily be split into several independent elementary tasks, we impose some simple programming rules. The main constraint is related to the high-level structure of the code corresponding to the application. This code has to be divided into two different parts: (1) an unique main task and (2) one or several elementary tasks. The functional activities that have to be performed are described within the elementary tasks. On the contrary the role of the main task consists mainly in initiating and coordinating the activations of these elementary tasks. This control activity is done using a set of three additional primitives called SUBMIT, WAIT and KILL. Once the input data needed to execute an elementary task is available (extracted either from the inputs provided to the global application by the user or from the result returned by a elementary task previously executed), the execution of the elementary task is submitted by the main task using the non-blocking primitive SUBMIT. The WAIT primitive allows to block the progress of the main task till all the mentioned elementary tasks have been completed. The WAIT primitive is necessary to create a synchronization point when two sets of elementary tasks have to be executed in sequence. The last primitive

[2]One of these portals is currently installed in our own institute at the following address: http://byzance.irisa.fr:1980/genogrid/

allows to stop the execution of the specified elementary tasks. The role of supervision played by the main task also includes the gathering of results returned by the elementary tasks and the final generation of a unique result file accessible from the Web portals. As indicated in [17] the computation performed by a main task can saturate the machine where this main task is executed. Thus in our approach both elementary tasks and main tasks are taken into account by the load balancing mechanism. In Section 5., we will describe how the main task and the elementary tasks submitted during the execution of this main task are allocated in the Grid.

4. A Multi-Level Organization

As indicated in the Introduction, the grid we consider is not a homogeneous and uniform set of resources managed by a single institution but rather a federation of several institutions that are located in different geographical areas and that agree to merge part of their computing facilities. A local administrator is associated to each domain and is in charge of managing the level of participation of its institution in terms of computing power. A domain is more than just an administrative entity. Resources within a domain are connected through local area networks. Therefore, a domain is a synchronous subnetwork in which bounds on the transmission delay exist and are known. These kind of assumption simplify the design of a master: a machine in charge of controlling the Grid's activities within its domain. The Grid is deployed over the internet. For security purpose, only a few machines in a domain are connected directly with the outside world. Therefore, interactions between domains can be limited to a group of machine, one per domain that are responsible to interact with the other domains. In a large scale Grid, the only reasonable assumption is to consider that this set of proxies corresponds to an asynchronous distributed system. An asynchronous system is characterized by the lack of a global synchronized clock, and puts no bound on the transmission delay of messages[3]. In some sense, domains can be considered as "synchronous islands in an asynchronous sea". Most of the difficulties encountered when designing Grid software are related to well known problems in distributed computing (observation of the global state of a distributed system, crash failure detection, ...) that are hard to solve when the system is asynchronous.

In the following sections, we describe successively the *domain level* and the *Grid level* on which our current organization of the Grid relies. Through the definition of a two-level architecture that can be extended to more than two levels, we aim to provide a simple and realistic model for Grid-computing. As resource allocation and dependability issues have to be tackled at the two levels, this model allows us to identify the nature of the potential changes in the computing environment, to determine how these changes are observed and to propose adaptive solutions.

4.1. The Domain Level

A domain is a set of heterogeneous nodes which communicate in a synchronous way. A node can be either a resource of the Grid or a machine devoted to control activities. The management of the domain is organized according to the master-slave model: in

[3]Such a bound may exist but either this value is unknown or the known value is so high that it cannot be used to define reasonable timeouts.

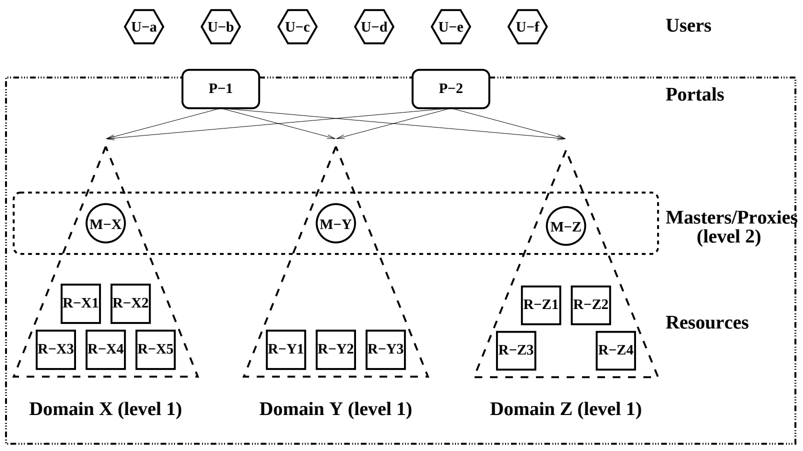


Figure 1. Grid Hierarchy

each domain, a single node named the *master* is selected to manage all the other nodes (named the *slaves*). In particular, the master has to schedule all the tasks carried out in its domain. At any time, the master can check the loads of its slaves. This information is used to compute an appropriate local scheduling of tasks. The composition of the domain is dynamic: the administrator of a domain can decide to add or to remove local resources from its local set of computing facilities accessible through the Grid. Of course, these modifications leads to increase or to decrease the computing capacity of the domain. We assume that resources always join or leave the domain by requesting to the master.

Nodes fail only by crashing. A faulty node behaves according to its specification until it stops prematurely and definitively its computation. As a domain is synchronous, all the crashes can be detected in a reliable way. When the crash of a resource is detected by the master, the master distributes again the tasks (previously allocated to the faulty node) among the remaining resources. The crash of the master has also to be tolerated. Some nodes (the *heirs*) are preselected to replace the master when it disappears. Thanks to a leader election protocol, a single heir is allowed to replace the previous master. If no node can replace the master, all the domain becomes unavailable. Of course, during the computation, the heirs have to keep track of the whole knowledge of their master. As the role of these backups is just to ensure that there is not a single point of failure per domain, we will no more discuss about them in the remaining sections.

4.2. The Grid Level

The Grid is an asynchronous network connecting different domains (Fig. 1). To avoid a flood of the Grid, only one node per domain is allowed to communicate with the other domains, this node is called the *proxy*. All the proxies of the Grid constitute a group. In practice, a single node per domain acts both as the proxy and the master. Like the composition of a domain, the composition of the network of domains is also dynamic. Through invocations of the *join* and *leave* operations, the local administrator of a domain can decide (independently from the other administrators) to add or remove his own domain from the Grid whenever he wants (maintenance and repair, alternating periods of private and public use of the local resources, ...). A domain is unavailable if no node of this domain can act as a proxy/master (occurrence of crash failures) or if the domain has been disconnected from the Grid (occurrence of communication failures, temporary partitions).

On one hand, join and leave operations are intentional and broadcast to all the members. On the other hand, evolutions caused by occurrences of failure are unpredictable and are not necessarily observed by all the members of the group. In the proposed solution, each proxy is coupled with a failure detector module which maintains a list of domains that it currently suspects to be unavailable. A *Group Membership* service will ensure that all the proxies, that are currently members of the group, share a consistent knowledge of the past history of the group, namely, the *join* and *leave* operations already executed and the failures suspected to have occurred.

5. Architecture of Paradis

The software architecture of Paradis (see Fig. 2) is defined according the two-level organization of the Grid. At every domain, the master has to manage the domain itself and the coordination with the other masters. These two distinct roles are played by two modules: **Domain Manager** and **Grid Manager**. The Domain Manager is in charge of managing resource allocation within a domain. When asked by the Grid Manager, it computes a score (also called a bid) for a task that reflects the adequacy between the domain and the task, i.e. the ability for the domain to quickly execute the task. The Grid Manager module is responsible for determining whether the domain it belongs to should execute or not a task; Therefore, it has to interact with the other grid manager modules. The Eden framework is in charge of these interactions; It provides the set of grid managers with a reliable group service that enables them to take fault-tolerant decisions for task allocation. Masters use the service of Eden to agree on a vector of scores (each entry in the vector corresponding to the score of a domain). Then, by applying a deterministic algorithm, every master can unambiguously determine on which domain the task has to be allocated. They select the same entry of the vector namely one entry which contains the best score that have been proposed and they decide that the corresponding domain will be in charge of the execution of the task. The two modules and the Web portal communicate via the exchange of notification events.

Section 5.1. (respectively 5.2.) discusses how the grid is managed at the domain level (resp. the grid level).

5.1. Management at the Domain Level

Every domain is managed following the *Master-Slaves* model, with a *Domain Manager* (DM) playing the role of the Master who assigns tasks to the resources of the domain, depending on their capabilities to execute them. The DM may crash, like any machine of the domain, so some *heirs* exist in the domain, ready to take the DM's place in case it crashes.

Bids and Auctions : The execution of a main task or the execution of a set of elementary tasks is asked through the generation of a request. All the tasks mentioned in the request will be executed within a single domain but perhaps by different resources of this domain. As the resources are different and have perhaps different workloads, the time required to execute a task may vary from one machine to another. A load balancing mechanism has to be used to find, at a given time, the best distribution of the requests on the resources that are currently available. In Paradis, this is implemented thanks to a bid mechanism. The goal is to determine if a given domain will be in charge of a request and to identify

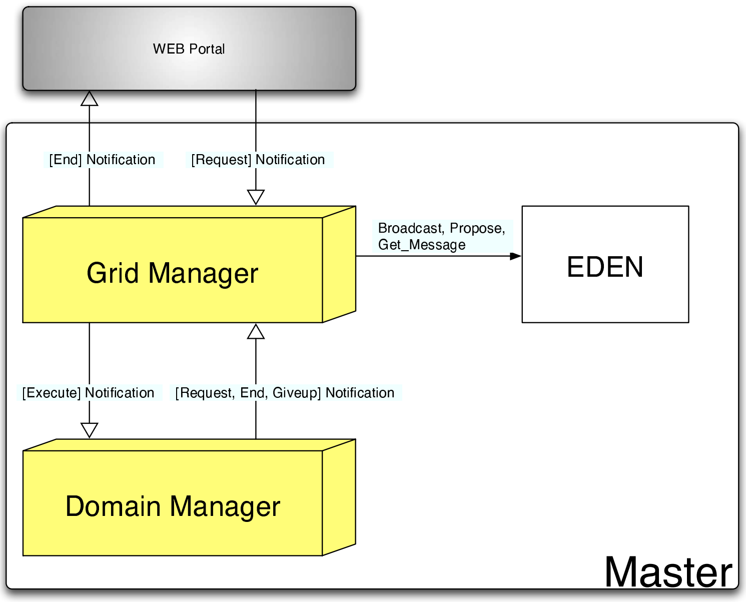


Figure 2. Architecture of Paradis

the local resources it plans to use. To achieve this goal, the request has been previously broadcast to all the DM using a atomic broadcast service.

First let us assume that a request R_i is composed of a single task. When a DM receives R_i from the GM, it determines which resource of its domain is the most appropriate to execute the task by computing, for every resource Res_j, a bid $bid_{i,j}$ (also called a score) representing the capability of Res_j to treat R_i as quickly as possible. Actually, this bid corresponds to the estimation of the time needed to complete R_i (waiting time before execution included). Thus it takes into account the current workload of a resource and the estimated execution time defined when the application has been registered. If R_i cannot be executed on Res_j for incompatibility reasons, then $bid_{i,j} = \infty$. Once the DM has computed the bids for all its resources, it will select the one Res_{win} with the lowest bid. If this bid is over a dynamic threshold (whose initial value is defined for each type of task and increases after each new computation of the bid of the task), the bid is also set to ∞.

When the request does not contain a single task to execute, but a bag of tasks, there are many ways to calculate the bids, depending on the strategy you want to implement. You may want to get the first result as soon as possible, or you may prefer to get the whole bunch of result as soon as possible. These two examples correspond to two different ways of compute the bids, and hence to two different task allocations. The main advantage offered by the bid mechanism is that the bid computation is totally independent from the architecture of the grid. To implement a new load balancing strategy, a user just has to change the formula that calculates the bid.

Once a bid has been computed, it is transmitted by the DM to the GM. At the grid level, the bids are used to make auctions between the different domains: thanks to agreement protocols, all the GM agree on a single vector of bids (one per domain of the Grid). The auction is won by the domain that has proposed the lowest bid (different from

∞). The use of a dynamic threshold allows to postpone the decision when the resources are already too busy.

Processing of a Request on a Domain: When a DM (which has proposed the lowest bid) receives some request R_i to treat from the Grid Manager (GM), it determines again which resource Res is the most appropriate to execute it. If Res is not available at this moment, then it adds the request to a list $Wait\_Req$ of requests to execute. Once Res is available, the DM executes the task T_i contained in R_i. Then it removes R_i from $Wait\_Req$ and add it to the list Exe_Req of the requests that are being executed. Once T_i is completed and has returned the result $result_i$, Res sends a message $\langle \text{END}, R_i, result_i \rangle$ to the DM. The DM removes then the request from Exe_Req and notifies the GM that R_i is completed and has returned the result $result_i$ thanks to the notification $\langle \text{END}, R_i, result_i \rangle$.

The task T_i contained in R_i may correspond to a main task. In this case, this main task generates some new requests R_{new} thanks to the SUBMIT function. Then, the DM notifies the GM that there is a new request R_{new} to send on the grid thanks to the notification $\langle \text{REQUEST}, R_i \rangle$.

In case there is no available resource to execute the task T_i contained in R_i (this may be the case if the resource that was supposed to execute it left the domain), the DM has to notify the GM that it cannot treat R_i, so that R_i can be executed on an other domain. This corresponds to the notification $\langle \text{GIVEUP}, R_i \rangle$.

5.2. Management at the Grid Level

At the grid level, due to the asynchronism of the system, grid management is more challenging than at the domain level. The lack of bounds on communication delays makes impossible to distinguish a slow proxy from a failed proxy. That prevents to implement reliable fault detection. Under such conditions, reaching an agreement on task allocation and grid composition is impossible. Hopefully, this result, know as the FLP impossibility result [6], can be circumvented thanks to the concept of unreliable failure detectors [3] that observe the availability of remote proxies. These failures detectors are said “unreliable” because, in an asynchronous system, the detection of a failed proxy by other proxies may be delayed or an available proxy can be mistaken for a faulty one by some proxy [3]. Unreliable failure detectors can be classified according to the properties (completeness and accuracy) they satisfy. A class of failure detectors denoted $\diamond\mathcal{S}$ is of particular interest because this class has been proved to be the weakest one enabling to solve a problem, called the Consensus problem, that is very close to the task allocation problem we have to solve.

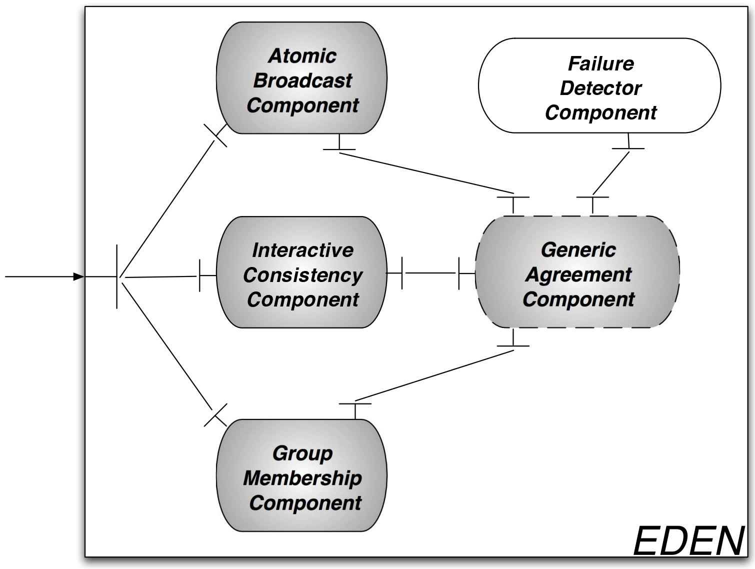
The consensus problem is defined in terms of two primitives called *propose* and *decide*. In the consensus problem, each process proposes an initial value and then executes a consensus algorithm until one of the proposed values is decided. The agreement problem we have to solve is close to the consensus problem. The difference with a classical consensus is that the decided value should not be one of the proposed values but a vector of the proposed values. This problem is quite similar to the Interactive Consistency problem.

EDEN : Eden (see Fig. 3) makes use of the unreliable failure detector concept to provide Paradis with a reliable group communication service [8]. Eden is based on a Generic Agreement Framework, called GAF, described in [10]. In GAF, different instantiations of the GAF parameters lead to generate different algorithms that solve efficiently the agreement problems. An instantiation is given by a concrete agreement component that implements the interface of the agreement service.

We identify three concrete agreement components which are:

- Atomic Broadcast. It ensures that messages sent to the group of proxies are delivered in the same order to all the members.
- Interactive Consistency. It ensures that all the members that propose a value decide a same vector of values.
- Group Membership. It is in charge of managing the computation and installation of new views whenever it is necessary. One important property of this service states that all members of the group should reach consensus about the current membership (who is in the group and who is not) [9].

Eden publishes a unified interface to the concrete agreement components needed by Paradis. This unified interface exports three operations: BROADCAST, PROPOSE and RECEIVE. The BROADCAST operation is used by a proxy to disseminate messages to the other proxies. It relies upon the service of the Atomic Broadcast component to ensure that every proxy will receive messages in the same order. The PROPOSE operation allows a proxy to propose a score for a given task. The Interactive Consistency component used to implement it ensures that every proxy will decide the same vector of scores. Finally, the RECEIVE method is the counterpart of the BROADCAST and PROPOSE operations; it ensures that every proxy will receive decisions on the vector and messages in the same order. The Group Membership component is used to provide a proxy with information about suspected remote proxies. A proxy gets this information through the RECEIVE operation of the Eden interface.



Atomic Broadcast Component
Failure Detector Component
Interactive Consistency Component
Generic Agreement Component
Group Membership Component
EDEN

Figure 3. Structure of the Eden framework

How Paradis uses this interface is explained in the following subsection.

5.3. Algorithm executed by the Grid Manager

As mentioned previously, the role of a Grid Manager (GM) is to manage the distribution of the tasks over the grid. It makes the junction between its own domain, represented by the Domain Manager (DM), and the other domains. Therefore, its activity is first, to communicate with the other GMs via Eden and with its coupled DM through notification events, and second, to manage two lists of tasks: a list of requests to allocate and a list of the requests that are currently being processed by every domain.

The Figure 4 presents the protocol executed by the GM. It consists of two parts: Actions in Part 1 are in response to messages received from the other GM through Eden; Actions in Part 2 follow notification events coming from the coupled DM.

Part 2 presents the 3 kinds of notification events a GM can receive from its local DM or from the local portal. When a request is submitted to the grid by a portal or by a DM, the GM is notified of this submission and broadcasts the request to all the GM through the BROADCAST function of EDEN. A notification of the end of the treatment of a request, or of the giving up of a request[4] received from the local DM is broadcast likewise.

Any message broadcast by a GM is received by every GM (sender included) via the Receive_Message() function of EDEN. This function returns the three types of messages broadcast by GMs (REQUEST, END, GIVEUP) in Part 1 and two additional messages types: DECIDE and REMOVE. Part 1 concerns the reactions to these messages.

- $\langle$REQUEST$, R\rangle$ informs the GM that there is a new request R to treat. It adds R to *Buffer*, a FIFO that contains all the requests that are not allocated yet (5).
- $\langle$DECIDE$, [bid_{i,0}, bid_{i,1}, ..., bid_{i,n}]\rangle$ returns the bids of all the domains for request R_i (6). The deterministic function Allocate() determines which domain D_{win} has proposed the best bid (7). The GM stores the information that R_i will be executed on D_{win} in the list $Allocations$ (8). If D_{win} corresponds to the domain of the GM, then the latest forwards R_i to its DM thanks to the function Execute() (9). Now that there is no current auction, a new one can be started, the GM is ready to bid (11).
- $\langle$END$, R_i, result_i\rangle$ informs the GM that the request R_i has been treated and returned the result $result_i$. Then, the GM removes R_i from the list $Allocations$ (14) and call the function Store_Result($result_i$) (15). This function will not "automatically" store the results: it will store it only if the request had been submitted to the grid by his local DM or the local portal.
- $\langle$GIVEUP$, R_i\rangle$ informs the GM that the domain that was in charge of request R_i has not been able to treat it. The GM removes R_i from the list $Allocations$ and adds it to the *Buffer* of requests to allocate.
- $\langle$REMOVE$, D_k\rangle$ informs the GM that the domain D_k just left the grid[5]. The GM removes then from the list $Allocations$ all the request that had been allocated to D_k (18-19) and add them to the *Buffer* of requests to allocate.

When there is no current auction being processed, a new one can start if there is some request to allocate in the buffer (26-27). In this case, the GM calls the function

[4]This may happen if the domain does not have any resource to treat the request any more.

[5]Either after having called the Leave() function of EDEN or having left without a warning. In this case, the leave of D_k have been detected by the Failure Detection module of Eden.

Grid Manager

Part 1

```
(1)  ready_to_bid ← True
(2)  While (True) do
(3)      $msg$ ← Receive_Message();
(4)      switch $msg$ :
(5)          case $\langle$REQUEST, $R\rangle$: Feed_Buffer($R$);
(6)          case $\langle$DECIDE, $[bid_{i,0}, bid_{i,1}, ..., bid_{i,n}]\rangle$: begin
(7)              $D_{win}$ ← Allocate ([$bid_{i,0}$, $bid_{i,1}$, ..., $bid_{i,n}$])
(8)              Store_Allocation ($D_{win}$, $R_i$);
(9)              if $D_{win}$ = MyDomain then Execute($R_i$)
(10)             endif;
(11)             ready_to_bid ← True;
(12)             end;
(13)         case $\langle$END, $R_i, result_i\rangle$: begin
(14)             Unstore_Allocation ($R_i$);
(15)             Store_Result ($R_i$, $result_i$);
(16)             end;
(17)         case $\langle$GIVEUP, $R_i\rangle$: begin
(18)             Unstore_Allocation ($R_i$);
(19)             Feed_Buffer(Request($R_i$))
(20)         case $\langle$REMOVE, $D_k\rangle$:
(21)             Foreach $R_l$ in Stored_Allocations($D_k$) do
(22)                 Unstore_Allocation ($R_l$);
(23)                 Feed_Buffer($R_l$);
(24)             done;
(25)     endswitch;
(26)     If ready_to_bid then
(27)         If ($R$ ← Read_Buffer() ) not NULL then
(28)             $bid_{i,MyDomain}$ ← Domain_Adequacy(MyDomain, $R_i$)
(29)             PROPOSE ($bid_{i,MyDomain}$)
(30)             ready_to_bid ← False
(31)         endif
(32)     endif
(33) done;
```

Part 2

```
(34) While (true) do
(35)     Upon notification of notif,
(36)         notif ∈ {$\langle$REQUEST, $R_i\rangle$, $\langle$END, $R_i, result_i\rangle$, $\langle$GIVEUP, $R_i\rangle$}
(37)     BROADCAST(notif);
(38) done;
```

Figure 4. Grid Manager's protocol

Domain_Adequacy(MyDomain, R_i) (28). This call makes the local DM compute the bid for the execution of R_i on the resources of the domain. The GM sends then this bid to Eden (29) and puts a lock on ready_to_bid to avoid concurrent bids (30).

In addition to this algorithm, the GM implements some protocols to synchronize the lists *Buffer* and $Allocations$ when it joins the grid.

6. Experimental Results

We have helped different teams of biologists working on genomic analysis to adapt the designs of their favorite applications to the few requirements imposed by our experimental Grid. More precisely, the codes of three different applications have been structured into main/elementary tasks and registered in the Grid in such a manner that they can now be executed by external authorized biologists from one of the portals (for example, from http://byzance.irisa.fr:1980/genogrid/). The first experiences we have conducted have validated the interest of a Grid approach for this range of applications that exhibit different characteristics. The two first applications aim to compare genomic sequences contained in two distinct data banks. In the first case, the application focuses on the study of viral infections of the testicles: from the raw data contained in the data banks, a reduced set of pertinent data is selected to be analyzed. Thanks to this initial filtering done once, the number of comparisons is approximatively equal to $16x10^9$. The second application aims to identify new human mitochondrial proteins [16]. Using a single machine, the execution of the first application lasts several hours while the second application ends after several months. In both cases, the computation power of a Grid allows to decrease the execution time: the main task is used to slice the whole computation into a pre-defined number of independent elementary tasks that are submitted in parallel. In each elementary task, a portion of the first data bank is compared to the whole second data bank. Interesting behaviors occur when the two applications are running concurrently. In the presence of applications whose duration are not of the same order of magnitude, the choice of a granularity for the elementary tasks (which has an impact on the number and the duration of the elementary tasks), the choice of adequate thresholds and the accuracy of the static estimation of the available resources offered by the Grid are essential factors when defining the static code of the main task. As the termination of the shortest application can be postponed after the end of the long-lasting ones, our experiences show that dynamic adaptation mechanisms have also to be added to the current software to maintain equity between the time-consuming activities and the light applications. The third application is related to the protein threading problem [14, 13]: it aims to assign a 3D structure to a protein sequence. This application requires three phases of computation that have to be executed sequentially. WAIT statements are used in the main task to create two intermediate synchronization points. In addition to the particular structure of its main task, this application is characterized by the fact that an elementary task generated during the last phase has an unpredictable execution time that ranges from 1 second up to one hour. Thus, estimations used to compute the bids are just average values. To face this uncertainty, the definition of an appropriate threshold and the possibility to go back on a previous decision (by generating a new allocation decision) reveal to be appropriate solutions.
Experiments done with several instances of the three above applications running at the same time validate the interest of a Grid and demonstrate also that the mixing of distinct applications requires to propose additional strategies: optimizing the use of the Grid resources does not imply that the satisfaction of the users is also always maximized. This problem is not specific to our approach and some of the proposed mechanisms (definition of a threshold, invalidation of a previous allocation) have shown to be efficient to implement complementary allocation policies.

As the main characteristic of our solution is to propose a hierarchical architecture where a set of processes (*i.e.* the proxies) interact only by executing agreement protocols, we now provide more details on the cost induced by the execution of agreement protocols. Within a domain, the interactions between a master and its local resources are based on a master/slave scheme and can be neglected compared to the cost of the agreement protocols executed by processes of different domains. Moreover these costs are not really specific to our solution. In the following, we focus on two particular scenarios. In the case of failure free scenarios, we consider the cost of the atomic broadcast service that is intensively used in our approach. To obtain a total order on the set of requests and to have a consistent observation of the progress of the computation, each proxy calls periodically this service.

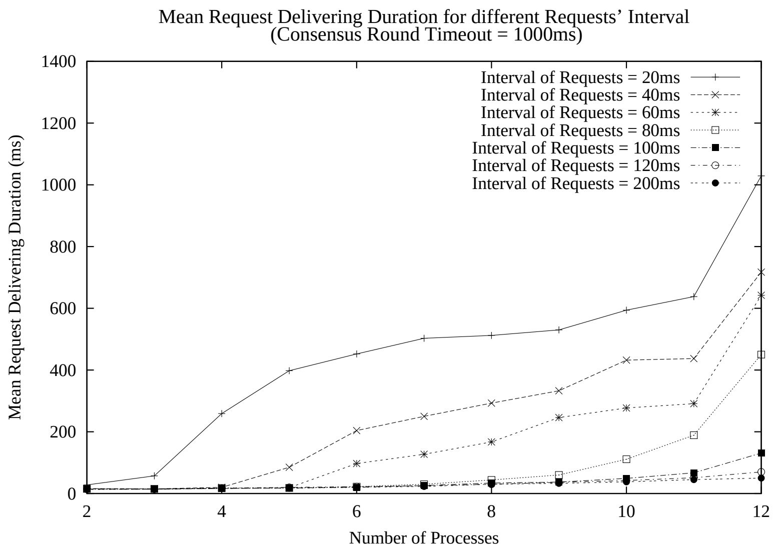


Figure 5. Cost of the Atomic Broadcast Service: failure free scenario

In Figure 5, we consider the mean time required to broadcast a request using an atomic broadcast service. It depends on the number of domains (number of processes) but also on the interval of time that elapses between two consecutive broadcasts. When the number of domains remains small (less than ten organizations), the cost of this agreement service is acceptable. Of course, consensus-based services are not scalable and such a technique cannot be used to federate thousands of domains. Finally the frequency of the requests is related to the granularity of the elementary tasks. A trade-off between the time required to allocate a task and the time required to execute this task has to be found. To circumvent this problem, elementary tasks can be aggregated in bunches of tasks that will be handled as single meta-tasks at the Grid level and then decomposed into elementary tasks at the domain level. Depending on their durations and their characteristics (known off-line), the number of tasks that compose one bunch can be adapted dynamically during the execution of the main task.

In Figure 6, we consider the cost induced by the occurrences of failures. More precisely, we consider the perturbations induced by a failure and the time required to install a new view. Due to the fact that the system is an asynchronous one, the failure detector mechanism is not reliable. To avoid the withdrawal from the Grid of an active domain, a rather long period of time is necessary before launching the computation of a new view that will eliminate a domain from the Grid. During this period, the suspected process (which is really crashed) is supposed to act as a coordinator from time to time. As a consensus algorithm based on the rotating paradigm is at the core of our solution, each round coordinated by a crashed process is useless and increases the execution time of the called service up to the fixed duration of a round. When the new view is computed and installed, state transfer mechanisms and synchronizations (required to guarantee the view synchrony property) create a last additional cost. In Figure 6, the cost of an atomic broadcast in a set of 5 domains is analyzed. A first crash of a proxy (without any local heir) occurs after 50 calls to this service. After 250 calls, a second proxy (in another domain) stops also definitely its activity. In both cases, the cost of a call increases each time the crashed process is supposed to act as a coordinator (5 times just after the first crash and 7 times just after the second crash). The view change has a high impact on the performances but immediately after a normal behavior is observed again. Note that timeouts have not be tuned to reduce this phenomena. Moreover crashes are rare and usually masked by the heirs of the proxy.

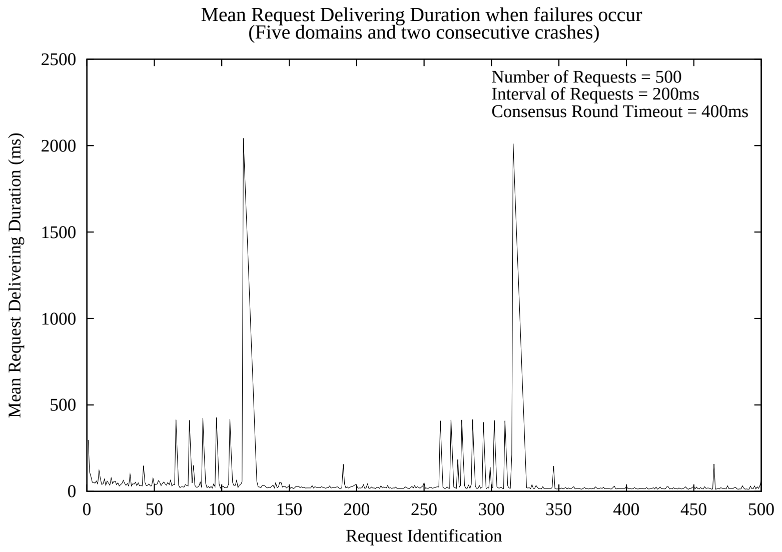


Figure 6. Cost of the Atomic Broadcast Service: crash failures of some proxies

7. Conclusion

This paper presents PARADIS, an adaptive system based on a Consensus building block that has been designed and implemented in a Grid dedicated to genomic applications. Resource allocation and dependability are particular issues that require a continuous adaptation to the changing computing environment. In the proposed approach, an agreement service is used by all the domains to acquire the same set of accurate data describing the current state of the Grid. Based on this unanimous observation, each domain can enact the right adaptation to react to the discovery of changes.

References

[1] J. Almond and M. Romberg, The unicore project: Uniform access to supercomputing over the web. *Proc. of the 40th Cray User Group Meeting*, 1998.

[2] D. P. Anderson, BOINC: A System for Public-Resource Computing and Storage. *Proc. of the 5th IEEE/ACM International Workshop on Grid Computing*, November 2004.

[3] T. Chandra and S. Toueg, Unreliable Failure Detectors for Reliable Distributed Systems. *JACM*, 43(2):225-267, 1996.

[4] W. Cirne and D. Paranhos and L. Costa and E. Santos-Neto and F. Brasileiro and J. Sauve and F. A. B. Silva and C. O. Barros and C. Silveira., Running Bag-of-Tasks Applications on Computational Grids: The MyGrid Approach. *Proc of the Int. Conference on Parallel Processing (ICPP)*, 2003.

[5] W. Cirne, F. Brasileiro, N. Andrade, L. Costa, A. Andrade, R. Novaes, M. Mowbray Labs of the World, Unite!!! *in Journal of Grid Computing (to appear)*, 2006.

[6] M.J. Fischer, N.A. Lynch. and M.S. Paterson, Impossibility of Distributed Consensus with One Faulty Process. *JACM*, 32(2):374-382, 1985.

[7] I. Foster and C. Kesselman, "The Globus Project": A Status Report. *Proc. of the 7th IEEE Heterogeneous Computing Workshop*, pp. 4–19, 1998.

[8] F. Greve, Réponses efficaces au besoin d'accord dans un groupe. *Ph.D. Thesis*, University of Rennes, november 2002.

[9] F. Greve, M. Hurfin, M. Raynal, and F. Tronel, Primary Component Asynchronous Group Membership as an Instance of Generic Agreement Framework. *Proc. of the 5th Int. Symposium on Autonomous Decentralized Systems* pages 93-100, 2001.

[10] M. Hurfin, R. Macêdo, M. Raynal, and F. Tronel, A General Framework to Solve Agreement Problems. *Proc. of the 18th IEEE Int. Symposium on Reliable Distributed Systems (SRDS'99)*, pages 56-65, 1999.

[11] M. Hurfin, J.-P. Le Narzul, J. Pley, and P. Raïpin Parvédy, A Fault-Tolerant Protocol for Resource Allocation in a Grid dedicated to Genomic Applications *Proc. of the 5th Int. Conference on Parallel Processing and Applied Mathematics, (PPAM 2003).*

[12] D. Lavenier, H. Leroy, M. Hurfin, R. Andonov, L. Mouchard, and F. Guinand, Le projet GénoGRID: une grille expérimentale pour la génomique. *Actes des 3èmes Journées Ouvertes Biologie Informatique Mathématiques*, pp. 27-31, France, 2002.

[13] A. Marin, J. Pothier, K. Zimmermann and J.-F. Gibrat, FROST: a filter-based fold recognition method. *Proteins*, 49(4), pp. 493-509, december 2002.

[14] J. Pley, R. Andonov, J.-F. Gibrat, A. Marin, and V. Poirriez, Parallélisations d'une méthode de reconnaissance de repliements de protéines (FROST). *Proc. of the 3th Journées Ouvertes de Biologie, Informatique et Mathématiques*, pp. 287-288, 2002.

[15] D. Powell, Special Issue on Group Communication. *CACM*, 39(4), 1996.

[16] Y. Tourmen, M. Ferre, Y. Malthiery, P. Dessen, and P. Reynier, Mitochondrial Diseases Preferentially Involve Proteins With Prokaryote Homologues. *in Comptes Rendus De L'académie Des Sciences-Biologies*, 327, pp. 1095-1101, 2004.

[17] P.K. Vargas and I. C. Dutra and V. D. do Nascimento and L. A. S. Santos and L. C. da Silva and C. F. R. Geyer and B. Schulze, Hierarchical Submission in a Grid Environment. *Proc. of the 3rd ACM Int. workshop on Middleware for grid computing*, December 2005, France.