A Self-Organising System for Resource Finding in Large-Scale Computational Grids

Fabrizio Messina, Giuseppe Pappalardo, Corrado Santoro
University of Catania
Dept. of Mathematics and Informatics
Viale Andrea Doria, 6 - 95125 - Catania (ITALY)
EMail: {messina, pappalardo, santoro}@dmi.unict.it

***Abstract*—This paper presents a novel approach for resource finding and job allocation in a computational Grid. The proposed system is called *HYGRA*, for *HYperspace-based Grid Resource Allocation*. The basic principle involves the construction of a virtual hyperspace in which the available amount of each resource type is used as a *geometric coordinate*, making each Grid node representable as a *point* in this virtual hyperspace. A distributed overlay construction algorithm aims at connecting each node with the *nearest* k *nodes* w.r.t. the *euclidean distance* defined in the hyperspace. In this system, a job request, which can be also represented as a point, navigates the hyperspace, from node to node, following the overlay links which minimize the euclidean distance between the current node and the target point representing the job itself. The paper describes the algorithms for overlay construction and resource finding and assesses their validity and performances by means of a simulation approach.**

I. Introduction

Computational Grids [3], [5] are computer systems whose infrastructure, in terms of software architecture and protocols, must feature very important characteristics, such as *fault-tolerance*, *scalability* and *efficiency*. These systems are made of thousands CPUs, therefore the operations provided must strongly take into account the size of the system. Such an aspect is particularly important for operations like job submission: this operation implies to find a Grid node featuring a desired amount of certain resources, such as number of CPUs, CPU time, RAM and disk space, network bandwidth, software libraries, etc.; when the overall system is composed of thousands of nodes, a proper *fast and efficient* node finding algorithm is mandatory, since a brute-force approach involving all nodes is obviously neither opportune nor sound.

A typical solution to the problem above involves the use of a *hierarchy of special repositories*, holding a distributed database which stores the information about each Grid node and its kind and amount of available resources. This is the technique employed by the *Monitoring and Discovery System* [9], [6], [1], provided by the Globus Toolkit[1], the de-facto standard software tool for building interoperable computational Grids. But even if MDS is widely used, it suffers of peculiar problems, well known to MDS designers, which seriously affect its performances: the most important problem is a possible *lack of consistency* of stored information, which is due to the unavoidable latencies between the instant in which a node changes its state (i.e. some resources become available or unavailable) and the instant in which the MDS database is updated; during this time period, the MDS database contains information which do not reflect the real node's state, thus affecting the validity of the resource finding phase; if the system size grows, in terms of number of Grid nodes, the probability of the occurrence of such an inconsistency increases.

A possible approach for solving the problem above, which has been studied in [4], [7], [2], foresees to directly query the nodes about their resource availability, instead of contacting "someone else" which could not have fresh information. Since querying *all nodes* is not a viable solution, the cited approaches consider the employment of *peer-to-peer* techniques, which aim at organising the nodes in a proper overlay network in order to make the query and match process easier.

With these concepts in mind, this paper proposes a P2P approach, called *HYGRA* (for *HYperspace Grid Resource Allocation*), strongly based on concepts proper of spatial computing [8]. The proposed solution, which is *completely decentralised* (no central or aggregated repositories, or super-peers), models the entire Grid as *n-dimensional hyperspace* in which resource types, with their availability, represent *coordinates*: each node thus virtually occupies a specific *point* in the hyperspace. Following this abstraction, nodes with resource availability close to each other feature *points*, in the hyperspace, that are near to each other; therefore, to find a node able to offer a certain amount of resources for a job implies to locate a specific region of the hyperspace and discover (one of or the best of) the nodes occupying that region.

To make this possible, HYGRA organises the Grid nodes in an *overlay network*, where each node is virtually connected to some other nodes and interactions among nodes can occur only following such links; overlay construction is performed by means of a decentralised algorithm in which each node/peer aims at ensuring that the node is always kept linked with *at most* k of the "nearest" nodes in the hyperspace.

The overlay network built so far is exploited by the resource finding algorithm which is thus reduced to a "geometric problem": given that we know the coordinates of the region holding candidate nodes (and this can be obtained from job's requirements), the query can start from *any node* and follow

[1] http://www.globus.org

the links leading to nodes with minimum distance from the target region. Also in this case, the approach is completely decentralised, since the peer receives the query, checks it and, if necessary, forwards it to a linked node until the target can be found.

The paper is structured as follows. Section II introduces the basic concepts and symbols used later on in the description of the HYGRA. Section III describes the system architecture. Section IV details the working scheme of HYGRA, reporting and explaining the overlay construction and resource finding algorithms. Section V discusses the result of a simulation study about the performances of HYGRA. Section VI concludes the paper.

II. Basic HYGRA Model

To better explain the technique proposed in this paper, we first introduce a mathematical formulation with a set of symbols and relations that are used to model the Grid environment and the hyperspace, and helps the understanding of the HYGRA algorithms.

A. Model of Resource and Job Request

We consider a Grid composed of n nodes; for each node i, let s_i denote its *state*, meant to describe resource availability on i, at a certain time instant; assuming that, in the overall Grid, all nodes offers m different types of resources (say, CPU, memory, disk space, network, OS type, etc.), the state s_i of a node will be a vector $(r_{i,1}, r_{i,2}, \ldots, r_{i,m})$; here each component $r_{i,j}$ represents the amount of resource j available at node i.

Let us define resource *domains* $D_1, D_2, \ldots, D_m$, meaning that $r_{i,j} \in D_j$ for $i=1 \ldots n, j=1 \ldots m$. If a resource is measured by a *quantity*, such as RAM, disk space, network bandwidth or CPU time, D_j will be a numeric interval, ranging from a minimum to the maximum admissible resource amount over all nodes. If the resource is of a different kind, such as the presence of a certain library or the availability of a certain library version, D_j will be a set of values each describing a potential resource instance.

In this system model, a job submission request, which carries job's requirements, is represented as the tuple $\overline{q} = ((f_1, q_1), (f_2, q_2), \ldots, (f_m, q_m))$, where q_j is the amount of the resource of type j requested by the job, and f_j is a *predicate* used to match the requirement with the availability: we say that node n^* can host a job carrying request $\overline{q}$ iff $f_j(r_{n^*,j}, q_j) = true, \forall j \in 1, \ldots, m$. We assume, without loss of generality, that predicate f_j can be either the $=$ or the $\geq$ relation: the former means that the request and availability must exactly match, while, in the latter case, the predicate succeeds if the resource availability is greater than the resource requested[2].

Let us introduce an example to better explain the practical usage of the symbols provided. We consider a Grid whose nodes possess the following resources: amount of RAM, number of CPU cores and GCC library version; let us suppose that there cannot be more than 8 GBytes RAM and 8 cores per node, and that some nodes offer glibc version 2.10.2 while others offer version 2.9.1. For this Grid, resource domains will be $D_1 = 0 \ldots 8192$ (assuming RAM is specified in Megabytes), $D_2 = 0 \ldots 8$ and $D_3 = \{2.9.1, 2.10.2\}$. In this system, a request for a job that needs 3 cores, at least 25 MBytes of RAM and glibc version 2.10.2 can be represented as $\{(\geq, 25), (=, 3), (=, 2.10.2)\}$.

[2]These two predicates cover most cases of job allocation requests in a Grid.

B. Model of the Hyperspace

The hyperspace model used by HYGRA starts from the formulation introduced above, provided that a transformation is applied to domains not featured by a numeric value: for each domain D_j which is a finite set of known values, we associate a (natural) number to each of such values; as instance, domain $D_3 = \{2.9.1, 2.10.2\}$ of the example above can be remapped to $D_3 = \{1, 2\} = [1, 2] \subset \mathbb{N}$.

After this transformation, each domain is indeed a subset of $\mathbb{N}$ or $\mathbb{R}$, therefore we can construct a *metric space* (S, d), where $S = D_1 \times D_2 \times \ldots \times D_m$, elements are vectors $v \in S$, and the metric d is the *euclidean distance*. Each Grid node i features a state s_i which (provided again the proper domain transformation) becomes a vector of the metric space $s_i \in S$: in other words, we can say that a node i of the Grid, according to its state, represents a *point* in the hyperspace S.

Following the same abstraction, in a job request $\overline{q} = ((f_1, q_1), (f_2, q_2), \ldots, (f_m, q_m))$, vector $q = (q_1, q_2 \ldots, q_m)$ is also an element of S, and $\overline{q}$, due to the presence of the predicates, determines a *partition* of S (or a *semispace*), $S(\overline{q}) \subset S$, made of all elements in S that satisfy predicates f_j:

$$S(\overline{q}) = \{v \in S : f_j(v_j, q_j) = true \quad \forall j \in 1, \ldots, m\}$$

We call $S(\overline{q})$ the *admissible region* for job $\overline{q}$ since any Grid node i such that $s_i \in S(\overline{q})$ is able to host the job. The aim of HYGRA is to provide a decentralised approach to allow the discovery of the nodes belonging to the admissible region of a given job that needs to be allocated and executed.

III. HYGRA Architecture

From the software architecture point of view, HYGRA is made by means of a multi-agent system. Since the overall structure is based an overlay network, each Grid node runs a NodeAgent holding three kind of information: *(i)* the *state* s_i of the node; *(ii)* the set of *references* of other NodeAgents, which belong to *directly linked nodes* of the overlay[3]; and *(iii)* the state s_j of each node j relevant to directly linked NodeAgents.

Each NodeAgent can communicate with other NodeAgents by exchanging the following types of messages:

[3]Of course such a reference may be an IP address, a couple IP/port, a FIPA agent name, etc. This is a matter of the implementation and its choice does not affect the architecture or the validity of the approach.

1) *Job Allocation Request* $\bar{q}$. This message is sent from a NODEAGENT, which is not able to allocate the job, to a linked NODEAGENT selected according to a proper forwarding policy detailed in Section IV.
2) *State Change Notification*. It is sent from the NODEAGENT of a node i to all the linked NODEAGENTs when node's state s_i changes to s_i' (due to a new job arrival or the termination of the execution of a running job); the message obviously carries information about the new state s_i'.
3) *2-hop Status Query*. This is a query message sent from the NODEAGENT of a node i to all its linked NODEAGENTs and aims at obtaining the state s_k of each node linked to all the nodes directly linked with i. A proper *2-hop Status Reply* message is expected following the transmission of this query.
4) *Link Creation Notification*. It is sent from a NODEAGENT i to a NODEAGENT j to inform the latter that i wants to be connected with j. After the reception of this message, NODEAGENT j updates its set of directly linked agents by including also i.
5) *Link Cut Notification*. It is sent from a NODEAGENT i to a (connected) NODEAGENT j to inform the latter that the link i/j has to be destroyed. After the reception of this message, NODEAGENT j updates its set of directly linked agents by removing i.

These messages are exchanged during the two main activities of the NODEAGENTs, *(i) overlay construction*, which aims at (self-)organising the overlay network in order to ensure certain neighborhood properties; and *(ii) job allocation*, in which a job request has to be fulfilled by checking the availability of resources of a node and, if this is not the case, properly forwarding the request to a linked NODEAGENT. The details and algorithms of such activities are described in the following Section.

IV. HYGRA WORKING SCHEME

As reported above, the working scheme of HYGRA is based on organising the Grid nodes in an overlay network featuring, in the metric space (S, d), a neighbourhood property based on the euclidean distance. In particular, for each node i, given the set $L(i)$ of the nodes directly connected with i, these nodes are those which feature the *minimal* euclidean distance $d(s_i, s_k), \forall k \in L(i)$. In other words, for each node i the following property holds:

$$\forall k \in L(i), \nexists s_h \in S, h \neq k, h \notin L(i) : d(s_i, s_h) < d(s_i, s_k) \quad (1)$$

If the property above holds, the resulting overlay network is a topological graph that can be traversed by means of e.g. a minimal path algorithm in order to find nodes belonging to the admissible region.

A. Construction of the Overlay Network

The overlay construction technique is based on an algorithm run by the NODEAGENTs which is regulated by two parameters, deg_{min} and deg_{max}, respectively the minimum and maximum degree of each node[4]; therefore $deg_{min}, deg_{max} \in \mathbb{N}, deg_{min} < deg_{max}$ and $deg_{min} \leq |L(i)| \leq deg_{max}, \forall i$. The basic algorithm run by the NODEAGENT of a generic node i can be summarised in the following steps:

Algorithm 1 Overlay Construction

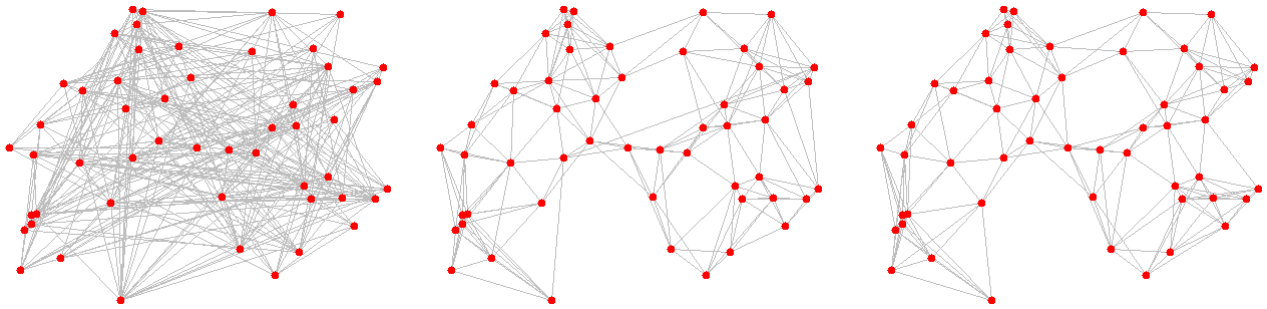
1) by sending *2-hop Status Messages* to each node of $L(i)$, build the set $L'(i) = (L(i) \cup (\cup_{\forall j \in L(i)} L(j))) - \{i\}$, that is the set of directly linked and 2-hop linked nodes of i;
2) since the NODEAGENT now knows all the states of nodes in $L'(i)$ (i.e. their *coordinates* in the hyperspace), order nodes in $L'(i)$ according to the distance to i (in ascendant order), i.e. $d(s_i, s_k), \forall k \in L'(i)$;
3) build the set $L^{''}(i)$ by taking *at most* the deg_{max} first nodes from $L'(i)$; these will be the nodes in $L'(i)$ which are *the nearest* to i;
4) if $L(i) = L^{''}(i)$, i is still connected to the nearest possible nodes, thus property (1) holds and the algorithm stops here.
5) connect node i with all nodes in $L^{''}(i)$; to this aim, disconnect, from i, nodes in $L(i) - L^{''}(i)$, by sending them a *Link Cut Notification* message and connect i with nodes in $L^{''}(i) - L(i)$, by sending them a *Link Creation Notification* message; then update $L(i) = L''(i)$.
6) restart from step 1.

The basic principle of the overlay construction algorithm should be quite clear: given any configuration of the overlay network, at each step of the algorithm each node tends to be connected to nodes that are nearer; this should be enough to ensure that, sooner or later, property (1) will be met. In such a case, condition in step 4 of algorithm 1 holds, meaning that the node has reached a stability; no more runs of the algorithm are needed unless the state s_i of the node changes due to the arrival of a new job or the termination of a running job: in this case, since the node has changed its coordinates in the hyperspace, property (1) could no more hold and the right links need to be re-created.

Bootstrapping the overlay network is also a simple operation: a new node x which wants to join must know only one existing node of the network, k: it has to link itself with k and nodes in $L(k)$, thus $L(x) = k \cup L(k)$, and immediately run Algorithm 1; at the first run, property (1) could not hold, but as soon as some steps of the construction algorithm are executed, a stable condition can be reached.

In order to understand the behaviour of the overlay construction technique, we built a software simulator, which is then described in Section V. Figure 1 shows some screenshots taken during the construction of the overlay following the algorithm described and using a Grid featuring two types of resources (in order to allow the representation of the

[4]According the literature on graphs, the degree of a node (or vertex) is the number of its links (edges) or its connected nodes.



(a) Initial condition (random) (b) After one step (c) Stable condition (after five steps)

Fig. 1. Overlay network construction from a random initial condition

network in two dimensions). The degree coefficients, deg_{min} and deg_{max}, are respectively set to 6 and 15. The initial condition of the network, in which all links are randomly set, is shown in Figure 1a, while Figure 1b and Figure 1c shows the network condition after respectively 1 and 5 iterations of Algorithm 1: the ability of the system to self-organise and to meet property (1) is quite evident.

Step 4 of Algorithm 1 reports a condition which, if met, ensures that the network, for what the single node is concerned, has reached a stability. The question is: can we ensure that sooner or later this condition will hold? Or, in other words, does the algorithm terminate? While, by looking the individual behaviour of the single node, at first sight the answer could be positive, the situation is much more complex when the *mutual influence* between linked nodes is considered. Indeed, since the *same link* connects two nodes, say i_1 and i_2, operations made by the algorithm in i_1 affect the behaviour of i_2 and vice-versa. If we take into account such a mutual influence, on the basis of the topological configuration of the nodes, during step 5 of Algorithm 1 the following situations may happen:

1) $i_2 \in L^{''}(i_1) \ \wedge deg(i_2) = deg_{max}$; according to the first condition, the algorithm in i_1 should connect i_1 to i_2, however, since the degree of i_2 is already deg_{max}, a new connection would cause an overcome of the maximum degree limit;
2) $i_2 \in L(i_1) - L^{''}(i_1) \ \wedge \ deg(i_2) = deg_{min}$; the link between i_1 and i_2 should be removed, but this operation would cause the degree of i_2 to go below deg_{min};
3) $i_2 \in L(i_1) - L^{''}(i_1) \ \wedge \ i_1 \in L^{''}(i_2)$; in this case, the same link is considered completely different by each node since it needs to be removed for i_1 but added for i_2; the result is a sort of "local oscillation" of the algorithm.

A simple check on the degree of each node in $L^{''}(i)$—resp. $L(i) - L^{''}(i)$—is able to easily solve situations 1 and 2: if the resulting degree is not between deg_{min} and deg_{max}, the node is not connected—resp. disconnected.

The third situation is more hard to tackle: indeed both nodes are in accordance with the algorithm and there is no way to choose if the link must be removed or preserved. It should be noted that such a local oscillation does not provoke a loss of consistency of the overlay network: indeed property (1) is not violated but the problem is only with a lack of efficiency due to a wasting of computational power since Algorithm 1 runs continuously without converging to a stable condition.

The solution we employed to avoid this problem is based on running a cycle of Algorithm 1 according to a certain *probability*; exploiting a model similar to that of simulated annealing, we associate to each node a *virtual temperature* t_i, which is initially set to a maximum value T_{max} and decreases at each cycle (till reaching 0) unless the node changes its state: in this case, t_i is reset again to T_{max}. Therefore, a cycle of Algorithm 1 runs with a probability $P_i = \frac{t_i}{T_{max}}$, thus ensuring that, even if the algorithm starts locally oscillating, sooner or later (if the node does not changes its state) this oscillation terminates and a stability is reached. According to this optimisation, the final behaviour of each NODEAGENT is described by the Algorithm 2 below:

Algorithm 2 Overlay Construction with Optimisation

1) Set $t_i = T_{max}$;
2) Compute $P_i = \frac{t_i}{T_{max}}$;
3) Run one cycle of Algorithm 1 with probability P_i;
4) if node i has changed its state s_i (due to job arrival or job termination), set $t_i = T_{max}$;
5) otherwise set $t_i = t_i - 1$, unless t_i is still 0;
6) go to step 2;

B. Resource Finding and Job Allocation Algorithm

The overlay construction algorithm described so far aims at organising the network in order to ease the resource finding phase. Indeed, the resource finding algorithm is quite simple and is based on *check-and-forward* policy: roughly speaking, once a node receives a job submission request, it checks if it can fulfill it (i.e. the node belongs to the admissible region), otherwise the node forwards the request to the linked node which is the nearest, according to its state, to the admissible region. The real algorithm is based on the principle above and applies some peculiar strategies for the choice of the next node (when more than one of it are candidates) and for the recovery when a path leading to a "dead end" (i.e. a wrong path which cannot lead to the admissible region) is followed.

A request, which is carried by a *Job Allocation Request* is represented by the tuple $(\overline{q}, P)$, where P is the ordered sequence of nodes visited till now. The request is submitted to

any NODEAGENT of the Grid with P initially set to empty; when a NODEAGENT receives such a message, the following algorithm is executed:

Algorithm 3 Resource Finding

1) on the arrival of a job request $(\overline{q}, P)$, check if the node can host the job, that is if $f_j(r_{i,j}, q_j) = true, \forall j \in 1, \ldots, m$;
2) if the condition is met, allocate the job in node i and terminate the algorithm with success;
3) if the previous condition is not met, build the set $L(i) - P$ and check if the set $N = (L(i) - P) \cup S(\overline{q})$ is not empty, i.e. determine the set of nodes in $L(i) - P$ which belong to the admissible region;
4) if such a set is empty, select a node i' in $L(i) - P$ which minimises the distance $d(s_{i'}, q)$;
5) if set N is not empty, select a node i' in N on the basis of an heuristic $H(N)$ which is detailed below;
6) if one of the previous two steps is successful, and thus node i' exists, we are approaching the admissible region, therefore update $P' = P \oplus \{i'\}$ by concatenating i' to sequence P, and forward the request $(\overline{q}, P')$ to node i';
7) if $L(i) - P = \emptyset$ there is no node that can allow the request to approach the admissible region since all linked nodes have been already visited and the algorithm would end in a infinite circular loop; in this case there are two possible causes: *(i)* the admissible region contains no nodes, or *(ii)* the path followed led to "local minimum". Indeed, with the current knowledge, the NODEAGENT has no way to understand the real cause and it can only *suppose* that the path is wrong: maybe making a different choice could help in finding the target, therefore we find, in the sequence P, the position of i and select the *previous node* i''. If such a node exists, the NODEAGENT forwards the request to i'', otherwise (that is, i is the *first* node in P) the algorithm terminates with a "node not found" message, meaning that the job request cannot be fulfilled.

If step 2 succeeds, the job has to be allocated on node i and the amount of resources needed by the job must be granted to it; in this case, the state of node i changes from s_i to s'_i, the node occupies another point in the hyperspace, it becomes "hot" and overlay construction algorithm restarts in order to let the network to re-organise itself. In a similar way, when a job terminates its execution on a node, it releases all the resources needed: also in this case the state of the node changes and a restart of the overlay construction algorithm is triggered.

Step 5 of Algorithm 3 entails to select the next node according to an heuristic; it is employed when, in proximity of the admissible region, the set $L(i) - P$ contains more than one node belonging to $S(\overline{q})$, so the question is *which one* of such nodes we have to select. To this aim, we have proposed and tested two different strategies:

1) ***MaxFree***, it selects the node that has the *highest amount* of free resources; in order words, the *further* node, with respect to $\overline{q}$, is chosen;
2) ***BestFit***, it selects the node that *best fits* the allocation, leaving the amount free resources nearer to zero; in other words, we choose the *nearest* node, with respect to $\overline{q}$.

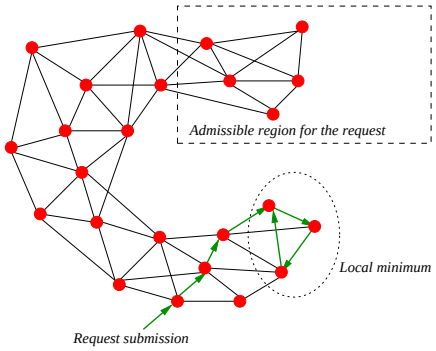


Fig. 2. A local minimum during resource finding

A final remark is needed to explain step 7. As it has been detailed in Algorithm 3, since there is no global knowledge or view of the network, there is no way, for a NODEAGENT, to understand if the admissible region is empty; the only fact which can be deducted is that the NODEAGENT is no more able to proceed further. However, as it is depicted in Figure 2, experimental results proved that such a condition occurs also in some extreme cases in which nodes are placed in points such that a path, for a certain job request, *seems* to lead to the admissible region, but indeed reaches a "dead end" or, in other words, what we call a *local minimum*. To solve such conditions, a second choice is given to the algorithm by *backtracing* to a previous node of the path followed: a different branch of the graph is selected thus increasing the probability to exit from the local minimum and reach the admissible region. Obviously, if the admissible region is really empty, all the alternative branches selected would end in nodes still visited and, sooner or later, the first node of the path will be reached again: after this, no choices will be left and therefore the algorithm will end with a failure indication (by high probability).

V. PERFORMANCE EVALUATION

A. The Simulator

In order to study the performance of HYGRA, we implemented a software simulator. It is a time-driven simulation tool which is able to represent the behaviour of Grid nodes, modeling both the overlay construction and the resource finding algorithm, also simulating job allocation and execution. The tool is capable to simulate the flow of time by means of discrete "ticks". For each timer tick **a)** a step of the *overlay construction algorithm* is executed on all nodes; **b)** a step of the *resource finding algorithm* is executed for all the requests circulating in the network; **c)** a bunch of *new job requests* is generated, on the basis of a frequency parameter specified in the configuration file (see below); **d)** a *check on the execution*

TABLE I
CRITICAL SIMULATION PARAMETERS

System Load	
-Parameter-	*-Admitted values-*
(a) Req.N.distribution	*Poisson* or *Random*
(b) Res.X.distribution	*Poisson* or *Random*
(c) Req.X.distribution	*Poisson* or *Random*
Algorithm Behaviour	
-Parameter-	*-Admitted values-*
(d) strategy	*Backtracking* or *Stop*
(e) node-selection	*ByID* or *ByCoordinates*

of the allocated job is performed, deallocating from the node terminated jobs.

The simulator takes various parameters as inputs and produces some indexes, briefly explained below, in order to evaluate the goodness of the proposed technique. Table I summarises a subset of the input parameters (the most important ones) that affect the average load of the system and the behaviour of the algorithm.

Parameter (a) specifies the probability distribution by which the number of requests per tick will be sampled, while parameter (b) specifies the distribution of the amount of resources over the overlay network[5]. Parameter (c) is very similar to the previous but specifies the distribution of the specific resource when a job submission request is generated. A well-tuning of the parameters (a-c) would induce different load conditions in the Grid and thus permits the evaluation of the approach on the basis of different system configurations. Parameter (d) is strictly bound to the behaviour of the resource finding algorithm. If set to "backtracking" the simulator is forced to adopt the backtracking strategy whenever a local minimum is found; on the other hand, by specifying "Stop" the simulator is forced to conclude the journey of the request whenever no choices are available[6].

It's worth observing that the backtracking strategy is generally affected by the dynamics of the overlay network, i.e., by the unexpected changing of coordinates and the subsequent reorganization of links. Indeed, once the request has reached a local minimum, one or more node previously saved in its history (the traversed path) could have changed its own coordinates due to a resource allocation or releasing. In this case, our choice was to end the journey of the request and signaling a failure, exactly as the totality of nodes in the path were visited back and no any alternative paths were found. We call this case the *hole-in-history* exception. However, we verified it occurs by very low probability, thus measuring and discussing it anymore it's probably not worth.

The parameter (e) is concerned about the discerning of the set of nodes already visited from those not yet traversed. When "ByCoordinates" is specified, the comparison of the nodes is based on their coordinates. In other words, the request stores not only the list of IDs of the visited nodes, but also the set of vectors corresponding to their coordinates. If one or more nodes are found in the set of neighbours such that their coordinates do not match with those stored in the history, such nodes are eligible for the selection even if they have been already visited. It is easy to understand that when "ByCoordinates" is set, there is no proof that the algorithm will terminate, i.e., the journey of the capsule will end. However, the termination of the algorithm could be trivially guaranteed by imposing a max value in the number of nodes visited by each request. The motivation behind the introduction of the variant "ByCoordinates" is still the dynamics of the overlay network. In fact, basing the comparison on the ID of nodes might be too restrictive because some nodes that have changed their coordinates, will have reorganised their links. Accordingly, they could take part in another path which might be useful to find the way to the admissible region.

B. Experiments and Results

We made a series of experiments on a test-bed of 100 nodes and two type of resources with a value ranging in the interval $[100, 200]$ by a uniform random distribution (see parameter (b) of Table I). Moreover, the value of deg_m and deg_M for the overlay construction are respectively set to 8 and 15, and job requests are generated using a Poisson distribution, with an average value ranging from 20 jobs/tick to 150 jobs/tick (parameter (a) of Table I). The values for resources requested (parameter (c) of Table I) by each job are sampled from the Poisson distribution using an average of 50. Job execution duration is also randomly generated with a Poisson distribution using an average value of 40 time ticks.

The results of the simulations are reported in Tables II and III. $NAlloc$ and $NFails$ represent, respectively, the number of requests successfully allocated, and the number of *failures* of the algorithm, i.e. a candidate node exists but it cannot be found. On the other hand, $NRej$ is the number of requests that cannot be allocated since there is no node able to support them[7]. Anyway we show for shortness only the ratio $\frac{NFails}{NAlloc}$, which we say $FailRatio$. The simulator also produces two other interesting indexes, $NSteps$ and $Optimality$ ($Opt.$ in Tables II and III). $NSteps$ represents the number of hops (nodes) performed, in average, by the allocation algorithm before a suitable node has been found. The $Optimality$ index let us to understand the "goodness" of the algorithm and is evaluated as follows. Each time the algorithm performs an allocation for a request $\overline{q}$, say n_{alloc} the node which has satisfied the request, the $Optimality$ is computed by using the formula $1 - \frac{d(n_{alloc}, n_{bestSelection})}{max(dist(n_i, n_j))}$, where $n_{bestSelection}$ is the best candidate node computed by doing *(i)* a total ordering of all nodes basing on the selected resource finding strategy ($BestFit$ or $MaxFree$), and finally *(ii)* selecting the best node from that ordering.

[5] X has to be replaced by the id of the resource

[6] The "Stop" variant is useful to assess the improvement provided by the introduction of the backtracking strategy.

[7] Clearly, $NAlloc + NRej = NReq$.

C. Evaluation of Results

Table II and III show the results of the simulation study; here we reported only the results of strategy $MaxFree$ because, for all the indexes evaluated, it behaves quite better than $BestFit$.

The first and most important remark to highlight is that the *number of failures* in Table II is lower than that reported in Table III, showing the advantages of applying the backtracking strategy. Furthermore, the overall trend of fail-ratio in Table III can even be considered low enough if compared to the total number of requests (see "FailsRatio" in Table III).

The second interesting parameter is the *average number of steps* performed by the allocation algorithm, which reasonably increases with the increment of the job generation rate. We have to remark also that the performances of the "Backtracking" strategy is comparable to that exploited by the "Stop" strategy in term of number of steps, in average, necessary to allocate a request.

The last performance parameter to take into account is the *Optimality*, that in our experiments appears to be independent of the job generation load[8].

Finally we have to report that even simulations were performed also for the "ByCoordinates" variants, we do not include the related results here, because the improvement in term of performance is not so significant.

TABLE II
STRATEGY: BACKTRACKING / NODE-SELECTION: BYID

Jobs/Tick (avg)	NFails	Fails Ratio	NSteps	Opt. (avg)	N.Req. (avg)
20	0	0.0	6.26	0.61	4972
30	7	0.0014	35	0.66	5043
40	8	0.0016	67.52	0.66	4951
50	6	0.0012	57.32	0.63	4982
60	8	0.0017	94	0.63	4732
70	8	0.0016	86	0.61	5043
80	19	0.0040	99	0.59	4721
100	24	0.0048	113	0.66	4996
120	32	0.0064	102	0.61	5023
150	34	0.0068	112	0.65	5012

TABLE III
STRATEGY: STOP / NODE-SELECTION: BYID

Jobs/Tick (avg)	Fails	Fails Ratio	NSteps	Opt. (avg)	N.Req. (avg)
20	0	0.0	5	0.68	4978
30	12	0.002	38	0.67	5079
40	32	0.006	59	0.62	4951
50	30	0.009	65	0.64	4982
60	43	0.008	86	0.63	4732
70	54	0.0016	78	0.61	5043
80	72	0.0040	72	0.59	4721
100	112	0.022	88	0.61	4996
120	124	0.025	82	0.58	5023
150	145	0.029	95	0.63	5012

[8]Therefore the algorithm, as for optimality, does not suffer of a performance degradation due to increasing load conditions.

VI. CONCLUSIONS

This paper has described a novel technique, based on a self-organising approach, for solving the problem of job allocation/resource finding in large scale computational Grids. The proposed system, called HYGRA, exploits spatial computing concepts and maps the entire Grid system into an hyperspace where each node, according to the availability of its resources, virtually occupies a *point* in the hyperspace. A completely decentralised algorithm, which exploits the *euclidean distance* among nodes, is able to self-organise an overlay network where each node is virtually linked with some other nodes feature a specific *neighbourhood property*. A *check-and-forward* algorithm is then employed during job submission to search, by surfing the overlay network, the node able to host at best the job, given its requirements. The proposed technique has been evaluated by means of software tool, able to simulate not only the behaviour of the algorithms but also the dynamics of job generation, submission and termination. Simulation results, provided in terms of computational cost, effectiveness and sensitivity with respect to Grid load conditions, have shown the validity of the HYGRA system.

REFERENCES

[1] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman, "Grid Information Services for Distributed Resource Sharing," in *10^{th} IEEE International Symposium on High-Performance Distributed Computing (HPDC 2001)*, San Francisco, August 6-9 2001.

[2] A. Di Stefano and C. Santoro, "A Peer-to-Peer Decentralized Strategy for Resource Management in Computational Grids," *Concurrency & Computation: Practice & Experience*, 2006.

[3] I. Foster and C. Kesselman, Eds., *The Grid (2nd Edition): Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2004.

[4] A. Iamnitchi and I. Foster, "On Fully Decentralized Resource Discovery in Grid Environments," in *International Workshop on Grid Computing 2001*, Denver, CO, Nov. 2001.

[5] D. Minoli, *A Networking Approach to Grid Computing*. Wiley InterScience, 2005.

[6] X. Z. J. Schopf, "Performance Analysis of the Globus Toolkit Monitoring and Discovery Service, MDS2," in *International Workshop on Middleware Performance (MP 2004) at IPCCC 2004*, April 2004.

[7] D. Talia and P. Trunfio, "Toward a Synergy Between P2P and Grids," *IEEE Internet Computing*, vol. 7, no. 4, pp. 94–96, 2003.

[8] F. Zambonelli and M. Mamei, "Spatial Computing: an Emerging Paradigm for Autonomic Computing and Communication," in *1st International Workshop on Autonomic Communication*, Berlin (Germany), October 2004.

[9] X. Zhang, J. Freschl, and J. Schopf, "A Performance Study of Monitoring and Information Services for Distributed Systems," in *12^{th} IEEE International Symposium on High-Performance Distributed Computing (HPDC 2003)*, Seattle, Washington, June 22-24 2003.