Journal of Computing and Information Technology - CIT 16, 2008, 4, 333–338
doi:10.2498/cit.1001402

HGRID: A Self-configuring Grid Resource Discovery

Antonia Gallardo[1], Luis Diaz de Cerio[2] and Kana Sanjeevan[1]

[1] Department of Computer Architecture, Polytechnic University of Catalonia, Barcelona, Spain
[2] Department of Automation and Computer Science, Public University of Navarre, Pamplona, Spain

Grid Resource Discovery Service is a fundamental problem that has been the focus of research in the recent past. We propose a scheme that presents essential characteristics for efficient, self-configuring and fault-tolerant resource discovery able to handle dynamic attributes such as memory capacity. Our approach consists of an overlay network with a hypercube topology connecting the grid nodes and a scalable, fault-tolerant, self-configuring search algorithm. By design, the algorithm improves the probability of reaching all working nodes in the system, even in the presence of non-alive nodes (inaccessible, crashed or nodes loaded by heavy traffic). We analyze the static resilience of the approach presented, which is the measure of how well the algorithm can discover resources without having to update the routing tables. The results show that the presented approach has significantly high static resilience.

Keywords: static resilience, fault-tolerant, grid resource discovery, hypercube, self-configuring protocol

1. Introduction

A Grid is an environment which allows the sharing of resources (computers, disk space, memory, bandwidth, data, instruments such as telescopes or microscopes etc.) that are connected to a network (e.g. the Internet).

The main objective of a Grid is to enable users to solve problems using the available collective resources. In this way, the Grid resource discovery service plays a fundamental role, allowing grid-enabled applications to locate resources based on a given set of requirements.

Resource search in Peer-to-Peer (P2P) systems offers an attractive approach to deploy fully distributed fault-tolerant Grid discovery service. But for the Grid, we have some extra requirements such as the existence of dynamic resource information (e.g. available memory, disk space, etc). In these cases, some of the P2P searching techniques are difficult to apply because they are more suitable for non-dynamic content.

Our architecture is based on an overlay network topology in the form of a hypercube which interconnects nodes provided by the Virtual Organizations (VOs). We also present a self-configuring resource discovery algorithm that is able to adapt to complex environments where some nodes might be non-alive (crashed, inaccessible, experiencing heavy traffic, etc.) when a resource is required.

Resilience in the presence of node failure has different aspects – *static resilience* and *routing recovery*. As the present work is focused on the resource discovery algorithm we only address *static resilience* in this paper – that is, how well our approach can locate required resources before routing tables are updated by the routing recovery algorithm in order to remove non-alive nodes in the overlay [4]. The other issue, *routing recovery*, is not addressed in this paper as this issue is related to the building and maintaining of the overlay topology.

The rest of the paper is organized as follows: In Section 2 we present an overview of our overlay network architecture. Section 3 describes our resource search algorithm. A brief overview of related work is presented in Section 4. In Section 5, the performance of the algorithm is evaluated. Conclusions and our plans for future work can be found in Section 6.

2. The Hypercube Overlay Architecture

In the hypercube overlay network that we present – named HGrid – each VO belonging to the Grid provides available resources (computers, applications, disk space, memory etc.) and makes them accessible through what we call the Grid Information System (GIS) [1].

In HGrid, the interconnections between GISs have the topology of a hypercube. An n-dimensional hypercube (H_n) has $V(H_n) = 2^n$ nodes, where each one represents a GIS. Each node (or GIS) has an identifier that goes from 0 to $2^n - 1$. Two nodes are said to be directly connected to each other (they are said to be neighbors in the m-th dimension) if the binary representations of their identifiers differ exactly by the m-th bit. Therefore, in a complete hypercube H_n, each vertex (GIS) has exactly n neighbors. Figure 1 illustrates the architecture for the interaction among 2, 4 and 8 nodes respectively.

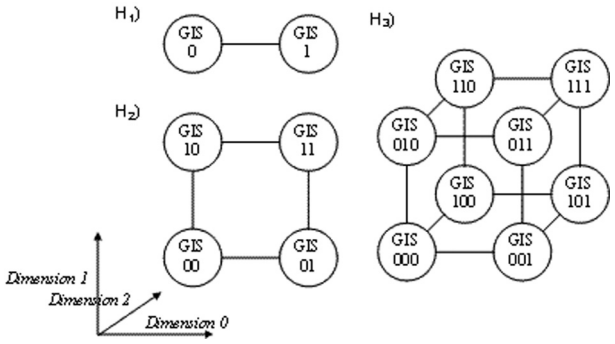

Figure 1. H_1) The architecture for the interaction among 2 nodes, a one-dimensional hypercube, H_2) 4 nodes, a two-dimensional hypercube and H_3) 8 nodes, a three-dimensional hypercube.

3. Resource Search Using HGRID

In this section we present a scalable self-configuring resource search algorithm (named Algorithm-H) that is able to adapt to complex environments.

It is possible to initiate a search request from any of the live nodes. For reasons of clarity however, the examples used from now on assume that node 0 is the start node, without any loss of generality.

3.1. The Search Procedure in an H_n Using Algorithm-H

The search procedure starts when a consumer wants to discover a Grid service. The consumer connects to one of the GIS nodes of the system and requests a service (a resource or some resources). The service discovery is tried first inside the requester's own GIS If there is no provider, then the request is redirected to other GISs. Figure 2 shows the pseudo-code that is performed at each node when a new service request message arrives:

1) When a new service request message is received by a node the function *processRequest(message.request)* is called. If the request included in the message cannot be satisfied, the node sets the value of *satisfyRequest* to false and the request is propagated.

Algorithm-H: Pseudo-code in a node when a new resource request message arrives

```
satisfyRequest = processRequest(message.request);
    IF (NOT satisfyRequest) THEN
        IF (startNode) THEN
            v_d = {0, 1, ..., n – 1};
            v_a = { };
        ELSE
            v_d = message.v_d;
            v_a = message.v_a;
        ENDIF
        v_d, d_alive, n_non-alive = statusNeighbors(v_d);
        IF (n_non-alive > 1) THEN
            v_a2 = addToList(v_a, d_alive);
        ENDIF
        FOR k=0 TO (v_d.size() – n_non-alive – 1) DO
            message.v_d = createList(k, v_d);
            IF (v_d [k] = d_alive ) THEN
                message.v_a = v_a2;
            ELSE
                message.v_a = v_a;
            ENDIF
            sendToNeighbor([v_d [k], message);
        ENDFOR
        FOR j=0 TO (v_a.size() – 1) DO
            IF (neighbor v_a [j] is not parent node)
                message.v_d = { };
                message.v_a = { };
                sendToNeighbor([v_a [j], message);
            ENDIF
        ENDFOR
    ENDIF
```

Figure 2. Algorithm-H pseudo-code.

Otherwise, *satisfyRequest* is set to true and no propagation is performed. The message forwarded is composed of the request (*message.request*) and two vectors of dimensions (*message.*v_d and *message.*v_a).

In case the request cannot be satisfied and the node that receives the message is the start node (*startNode* is *true*), the list v_d is initialized to $v_d = \{0, 1, \ldots, n-1\}$ (the complete set of dimensions) and v_a is initialized to $v_a = \{\}$ (an empty list). Otherwise, v_d and v_a are initialized to the lists received along with the request message. In both cases, the lists represent the set of dimensions (neighbors) along which the message must be propagated.

2) The node calls the *statusNeighbors*(v_d) function and reorders the list v_d in such a way that the dimensions corresponding to the non-alive neighbors are located at the last positions of the list. For example if $v_d = \{0, 1, 2, 3\}$ and the neighbor along dimension 1 is not responding, then v_d is reordered to $\{0, 2, 3, 1\}$. The *statusNeighbors*(v_d) also returns two integer values $n_{non-alive}$ and d_{alive}. The integer value $n_{non-alive}$ represents the number of non-alive nodes in the reordered list v_d. The integer value d_{alive} represents the dimension of the last alive neighbor stored in v_d. For example, if $v_d = \{2, 1, 0, 3\}$ and its neighbors in dimensions 0 and 3 are non-alive nodes, $n_{non-alive} = 2$ and $d_{alive} = 1$.

3) If the number of non-alive neighbors ($n_{non-alive}$) is more than one, the node calls the *addToList*(v_a, d_{alive}) function. This function appends d_{alive} to the end of the list v_a and returns the new list (v_{a2}).

4) For each position k in *the* list v_d that represents a live neighbor node, the node calls the *createList*(k, v_d) function which creates a new list composed of all the dimensions located after position k in the ordered list v_d. In other words, if the number of elements in v_d (v_d*.size()*) is q, the function returns $\{v_d[k+1], \ldots, v_d[q-1]\}$. For example, if $v_d = \{0, 2, 3, 1\}$ and $k = 1$, the call to *createList*(k, v_d) will return $\{3, 1\}$. Also, for each alive neighbor, the v_a list is initialized. The request, v_d, and v_a are sent to the corresponding neighbor in the $v_d[k]$ dimension inside a new message by calling the *sendToNeigbor*($v_d[k]$, *message*) function. See Figure 3 (a complete example using Algorithm-H) where the start node (0000) sends $v_d = \{2, 1, 0\}$ and $v_a = \{3\}$ to its last alive neighbor (the only one in this case).

5) Finally, the node propagates the request to each of the neighbors along v_a dimensions only if the corresponding neighbor is not its parent node. The request travels inside a message together with v_a and v_d as empty lists.

Propagating the requests in this way, the effect of non-alive nodes is reduced. Making the rearrangement in the v_d list, non-alive nodes would propagate the request to fewer neighbors than alive ones (in case the propagation were tried). Consequently, the algorithm tries to isolate the nodes that are in a non-alive state so that they become leaf nodes (if it is possible). If, under the circumstances, each node has only one non-alive neighbor, then all live nodes can be reached. On the other hand, the nodes that are unreachable because of inaccessible or crashed nodes along a path to them, can be reached eventually via other nodes – using the v_a list.

3.2. A Complete Example Using Algorithm-H

Figure 3 illustrates a complete example. We transform the hypercube representation to that of a *tree-like* figure in order to illustrate better our search procedure (for example, some 'child' nodes could appear more than once during subsequent time steps).

A request for service P starts at node 0000 in a four-dimensional hypercube. We assume that none of the nodes has the service requested (note that this is the worst case). In the example, the value of the list v_d at the start node is $\{0, 1, 2, 3\}$ and the ordering after calling the *statusNeighbors()* function is $\{3, 2, 1, 0\}$. In this case 2, 1 and 0 are located at the last three positions of $v_d = \{3, 2, 1, 0\}$ because we assume that neighbors in dimensions 2 (0100), 1 (0010) and 0 (0001) are non-alive nodes. The neighbor in dimension 3 (1000) is the last alive node, so $v_{a2} = \{3\}$ and $v_a = \{\}$.

In five steps almost all of the live nodes inside the four-dimensional hypercube are visited

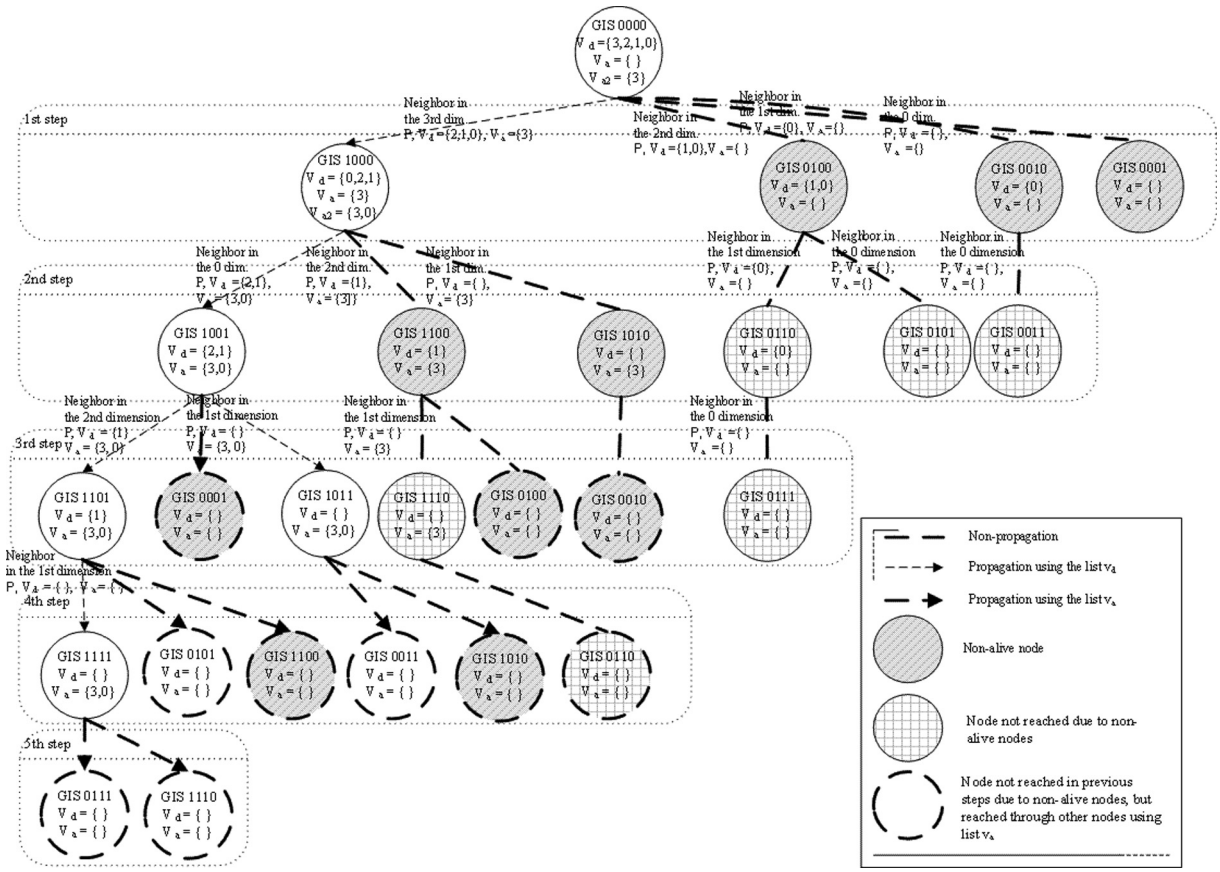

Figure 3. A complete example using Algorithm-H. A request of resource *P* started at node 0000 in a four-dimensional hypercube.

(all except node 0110), even when three out of four of the start node's neighbors (0100, 0010 and 0001) and two additional nodes (1100 and 1010) are presumed to be non-alive.

4. Related Work

The Globus Toolkit's Monitoring and Discovery System (MDS) defines and implements mechanisms for service and resource discovery and monitoring in distributed Grid environments [2]. Motivated by these issues, recently there have been several studies using the P2P model to build a decentralized architecture of VOs. Most of them adopt Distributed Hash Tables (DHTs), and a few of them introduce unstructured P2P topologies. All these studies indicate that some P2P models could help to overcome the challenges posed by the dynamic environment in Grids.

Adriana Iamnitchi and Ian Foster [6] suggest a decentralized architecture similar to the Gnutella P2P system. This approach is not able to guarantee that some required information existing in the system can be found, even when the system has no failures (none of the nodes are inaccessible or crashed). Moreover, a peer could be often reached several times by the same request. Our approach assures that nodes are reached only once. In the absence of failures, all nodes in the system are reached and if failures occur almost all the nodes are reached.

DHT-based systems handle unexpected node failure through redundancy in the network and some of them also do node lookups asynchronously or periodically, to compensate for disappeared nodes – for example, Kadmelia [7].

In order to enable efficient searches, a DHT needs to have the data-item distributed across the peers. Our approach does not require distributing the data-items, but each request sends from 0 to $N - 1$ messages. Keeping the state of highly dynamic data-items updated (such as available memory or CPU processing) requires sending a very large amount of messages in DHTs.

In HGrid, changes in the overlay network when a Grid node joins or leaves the system do not cause resource information (data-items) to be remapped, whereas in traditional DHTs, it causes both routing tables and data-items to be remapped.

Recently, an unstructured topology based on hypercubes has been proposed for use on Data Grids [5]. The nodes in this work contain pointers to shared data. Data Grids need to improve locality among distributed data (which are stored as pointers in the overlay nodes). In order to improve the locality of data, the paper imposes a hypercube topology of GIS (named DGIS). It then proposes a transposition algorithm to optimize the overlay network's topology according to the access statistics between peers (that is, to improve the data locality). However, the algorithm shown does not address non-alive nodes.

5. Performance Evaluation

Next we present simulations to evaluate if some required information that exists in the system can be found (lookup guarantees) without resorting to active recovery algorithms.

For this simulation, we have tested static resilience [2] with ten thousand, one hundred thousand and one million nodes - using 14, 17 and 20 dimensional hypercube overlays. All nodes have the same probability P_f of failure. P_f can be seen as the percentage of non-alive nodes that can be found in the overlay. We run the simulation for values of P_f between 0 and 50% – because we assume that the Grid environment is not extremely transient. Given a P_f, we start a request for service P at node 0, assuming that none of the nodes has the service requested, and count how many live nodes are not reached by the request P (failed paths). The simulation is repeated 20 times for each P_f getting the average number of failed paths. Finally we compare our approach (Algorithm-H) with two other search algorithms for hypercube overlays [5][3].

The average percentage of failed paths for varying P_f is shown in Figure 4. Our proposal offers substantially better static resilience than the HaoRen et al.'s algorithm [5]. The HGrid algorithm-H works very well in non-extremely transient environments, where a reduced fraction of the nodes are down at any given time ($< 50\%$ of failed nodes in the entire Grid). At the same time, Algorithm-H offers better resilience than Algorithm-P [3] as it can reach more live nodes without using active recovery algorithms.

Algorithm-H is not only scalable in overlay hops (n time steps where $n$is the dimensionality), but as seen in Figure 4, it is scalable in the number of nodes (2^{14} to 2^{20} from Figure 4).

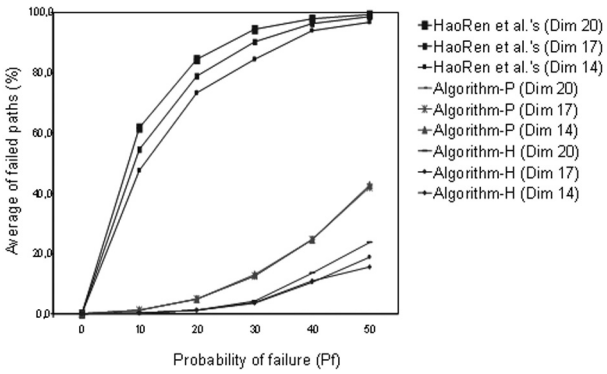

Figure 4. Percentage of average of failed paths for varying P_f across different search algorithms. HaoRen et al.'s algorithm is a non fault-tolerant algorithm described in [5]. Algorithm-P is a fault-tolerant algorithm described in [3].

6. Conclusions and Future Work

The proposed algorithm is scalable in terms of time, because it keeps the maximum number of time steps required to resolve a resource request, to a logarithmic scale with respect to the total number of nodes. Moreover, each node has only partial knowledge of the overlay Grid. Nodes do not require having the state information of the rest of nodes in the overlay, but only of the state of its neighbors. Therefore, our approach is also scalable in terms of data storage. Furthermore, scalability is also maintained by querying each node only once at the most (if possible). This important property (scalability) also extends to the number of nodes in the Grid – as can be seen clearly in Figure 4.

Unlike traditional approaches based on DHTs, our scheme is suitable for efficiently handling dynamic attributes such as memory capacity, without generating overhead across geographically distributed nodes.

Our method helps to balance the system load and is more efficient than other schemes like

flooding. Additionally, it is self-configuring when there are crashed or heavily loaded nodes.

By using the deep multi-dimensional interconnection of a hypercube, we provide enough connectivity so that resource requests can always be propagated in spite of non-alive nodes. This makes our proposed algorithm much more fault-tolerant when it is compared with other topologies such as centralized, hierarchical or trees.

In the absence of non-alive nodes, it is able to offer lookup guarantees.

Completing the comparison with DHT-enabled implementations is a goal for future work.

There are several interesting areas that have opened up as a result of this work and we are presently working on them: a) Design and evaluation of new request forwarding strategies, b) Incorporation of topology construction and maintenance algorithms and c) Evaluation in terms of response time, scalability etc. by simulation.

7. Acknowledgments

This work was supported by the Ministry of Science and Technology of Spain and the European Union under the references TIN2007-68050-C03-01 and TIN2007-68050-C03-02.

References

[1] R. BUYYA AND M. MURSHED, "GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing", *Concurrency and Computation: Practice and Experience (CCPE)*, Volume 14, Issue 13-15, (2002).

[2] K. CZAJKOWSKI, S. FITZGERALD, I. FOSTER AND C. KESSELMAN, "Grid Information Services for Distributed Resource Sharing". 10th IEEE International Symposium on High Performance Distributed Computing, *IEEE Press*, 2001.

[3] A. GALLARDO, L. DÍAZ DE CERIO, K. SANJEEVAN AND L. C. E. BONA, "HGRID: A Hypercube-Based Grid Resource Discovery", submitted to Special Issue on Grid Resource Management, *IEEE Systems Journal*, September 2007.

[4] K. GUMMADI ET. AL, "The Impact of DHT Routing Geometry on Resilience and Proximity", in *Proc. of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Karlsruhe, Germany, August 2003, pp. 381 – 394, ISBN:1-58113-735-4.

[5] HAO REN, ZHIYING WANG, ZHONGLIU, "A Hypercube based P2P Information Service for Data Grid", *Fifth International Conference on Grid and Cooperative Computing*, 2006, pp. 508-513.

[6] A. IAMNITCHI AND I. FOSTER, "On fully decentralized resource discovery in grid environments", *Second International Workshop on Grid Computing*, pp 51-62, London, UK, 2001. Springer-Verlag.

[7] KADEMLIA: A DESIGN SPECIFICATION, available-http://xlattice.sourceforge.net/components/protocol/kademlia/specs.html.

Received: June, 2008
Accepted: September, 2008

Contact addresses:

Antonia Gallardo
Departament de Arquitectura
de Computadors at the Universitat
Politécnica de Catalunya
Avda. del Canal Olímpic s/n
08860 Castelldefels
Barcelona, Spain
e-mail: agallard@ac.upc.edu

Luis Diaz de Cerio
Dpto. Auotmática y Computación
Universidad Pública de Navarra
Campus de Arrosadia
Edificio departamental de los Pinos
31006 Pamplona, Spain
e-mail: luismanuel.diazdecerio@unavarra.es

K. Sanjeevan
Departament de Arquitectura
de Computadors at the Universitat
Politécnica de Catalunya
Avda. del Canal Olímpic s/n
08860 Castelldefels
Barcelona, Spain
e-mail: sanji@ac.upc.edu

ANTONIA GALLARDO received the degree in telecommunications engineering in 2000 from the Universitat Politècnica de Catalunya (UPC) in Barcelona, Spain. Currently, she is a PhD student and assistant professor in the Computer Architecture Department at the Universitat Politècnica de Catalunya. Her research interests focus on distributed computing and grid systems.

LUIS M. DIAZ DE CERIO received the degree in telecommunications engineering in 1993 and the PhD degree in telecommunications engineering in 1998, both from the Universitat Politècnica de Catalunya (UPC) in Barcelona, Spain. He is currently an associate professor in the Automatics and Computing Department at the Universidad Pública de Navarra (UPNA). His research interests focus on distributed computing and grid systems.

KANA SANJEEVAN has a Bachelors degree from the Indian Institute of Technology, Chennai. He also has double Masters degrees in engineering and computer science from the University of California. He is currently an assistant professor in the Department of Computer Architecture, at the Universitat Politècnica de Catalunya (UPC) in Barcelona, Spain. His research interests focus on distributed computing and grid systems.