A random walk topology management solution for Grid

Cyril Rabat, Alain Bui, and Olivier Flauzac

Université de Reims Champagne-Ardenne,
BP 1039, F-51687 Reims Cedex 2, France
{cyril.rabat,olivier.flauzac,alain.bui}@univ-reims.fr

Abstract. GRID computing is a more and more attractive approach. Its aim is to gather and to share the resources of a network like the content, the storage or CPU cycles. A computational distributed system like *SETI@home* produces a power up to 70 TFlops whereas the current best parallel supercomputer *BlueGene* produces a power of 140 TFlops. Such a supercomputer costs very much contrary to a system like *SETI@home*. But the use of many computers to increase the global computational power involves several communication problems. We must maintain the GRID communication in order to make any type of computation even though the network is volatile.
In this paper, we present a model to represent GRID applications and networks in order to show faults impacts. We present a fully distributed solution based on a random walk to manage the topology of the GRID. No virtual structure needs to be maintained and this solution works on asynchronous networks. We also present some simulations of our solution.

1 Introduction

GRID computing is used to manage resources sharing and to gather resources over a network. We can distinguish two kinds of GRID:

- *High Performance GRID Computing (HPGC)* is composed by few supercomputers gathered in a cluster. During a computation, we cannot disconnect any computer. When a fault occurs, the computation stops and we must restart it. But this kind of GRID has two main advantages: all the results are true and we do not need topology management.
- *Desktop GRID* is composed of many nodes. Connections and disconnections from the GRID can be numerous. Often, no authentication is used on nodes and results can be wrong or corrupted. So even if no failure occurs on the network, we must check all the results. There is no fixed architecture and we need to manage the GRID topology.

In this paper, we present a new GRID topology management. This solution is based on a random walk and is tolerant to nodes disconnections. To describe it, we introduce an original network and application model for GRID. We use several layers to distinguish communication graph, routing protocol, messages

exchanges and GRID application.

In the next section, we present an overview of current GRID application types and particularly we focus on their architectures. Then we present our 5-layer model with a description for each one. Finally we show our random walk based solution with fault management and in the last section, we discuss on some simulations using this solution.

2 Related works

2.1 An overview of current GRID applications architectures

For several years, as claimed in [12], GRID computing has been the unavoidable solution for resources sharing: computation power, data, storage and applications. Demands for these kinds of resources grow exponentially but, on the other hand, many resources are unused. GRID computing enables us to federate them. Many solutions have been designed. We can gather them in three main categories:

- *Applications* designed to solve specific problems. In this category, we find the first version of the SETI@home project[1] developed to Search for Extra Terrestrial Intelligence ([3]).
- *Protocols* and *libraries* providing tools to develop GRID applications like JXTA ([14]).
- *Middlewares* offering a set of different services like resources discovery and monitoring, topology management... We find the following solutions: Xtrem-Web ([6]), Globus ([1, 11]), Diet ([5]), CONFIIT([10, 13]) or BOINC ([2]).

Whatever the category, several problems occur and particularly with the topology management. In order to provide a QoS over the GRID, to gather with its management and integration of new resources, it is necessary to define a strategy depending on the chosen centralization degree. Based on this criterion, we can classify the topology management of GRID applications into three categories: centralized, distributed and hybrid (semi-distributed).

Centralized topology management. This is based on pure client-server model: there is only one server on which all clients are connected. Submitters send requests to the server that distributes it to clients. Results are collected and returned to the submitter.

SETI@home ([3, 2]) for example, uses this kind of management: the data to be analyzed are centralized on the Berkeley server. Based on free cooperation, users download a client application that automatically contacts the server and downloads computation parameters. Results are collected by the server and saved into a large database. Links between clients and server are not critical: connections and disconnections are numerous (up to 4 million volatile clients) and each task is computed several times (2 to 3). The first reason is to avoid computational errors or malicious computation with comparison between results

[1] http://setiathome.ssl.berkeley.edu/

and the second one is to manage clients' failures.

This kind of management involves the overload of servers and a critical access: if the server crashes, the GRID fails to work.

Hybrid topology management. This kind of management establishes a hierarchy between nodes. Several interconnected layers of servers are substituted for the server. Nodes are managed by the lowest server and offer resources: storage, data, applications or/and computational power.

DIET [5] uses this architecture: it is composed of several interconnected servers called master agents (MA) in order to limit breakdown effects. If one of them does not respond, clients can contact another one. Under MAs, we have several layers of agents and at the bottom, the leader agents (LA) communicate directly to nodes. MAs are used to federate the GRID and to receive and diffuse requests to lower agents. LAs have a local knowledge of their associated resources. Upon job submission, the submitter contacts an MA that diffuses its request to intermediate agents and so on, down to nodes. If one of them suits this request, it reports its local agent and the response goes up to the MA and then to the submitter.

With this management, we decrease the overload of servers and we increase the breakdown tolerance. But it is difficult to establish a good hierarchy that has to suit the network topology as much as possible. Moreover, it overloads special management nodes and the breakdown of one of them involves GRID disconnection of the lower agents and their associated resources.

Distributed topology management. For this management, no hierarchy between nodes of the GRID is needed. There is no server: each node has the same purpose and is called a *servent*. It provides client and server tasks: computation, local resources management and communication.

For example, we find the fully distributed middleware CONFIIT ([10, 13]). Each node is set up into a virtual ring. When a neighbor crashes, nodes contact their next neighbors and so on. A token maintains the ring topology and another one manages the computation.

Although it is fully distributed, a ring structure is needed and it must always be maintained. This management uses too much computational power. Furthermore, a fixed structure like a ring or a tree involves default paths. They are more sensitive to breakdowns and each breakdown has more impacts.

2.2 Random walk

A random walk is defined by a sequence of vertices visited by a token that starts at node i and visits other vertices according to the following rule: when the token is at node i at time t, at time $t+1$, it is on one of the neighbors of i. This neighbor is chosen uniformly at random among all of them and probability to reach one of the neighbors is exactly $1/deg(i)$, where $deg(i)$ is the number of neighbors. In [15], the author shows us a survey on random walks.

A random walk has some quantities to evaluate its efficiency:

- the *cover time* C is the average time to visit all nodes of the graph.
- the *hitting time* denoted by $h_{i,j}$ is the average time to reach a node j for the first time starting from a given node i.
- the *return time* is the average time for a random walk to return to its original node. It is the special hitting time $h_{i,i}$.

Each quantity has a lower and upper bound as proved by U. Feige in [7, 8].

Random walks are used in several cases. In [9], the author gives an algorithm to compute routing tables of the shortest paths. But we can also find GRID applications. In [4], the authors show a method to distribute the tasks of a problem into a computational GRID in bounded time and in [16], an algorithm based on multiple random walks enables us to decrease the number of messages when querying in peer-to-peer networks.

3 Preliminaries

3.1 Model

To adapt our random solution to a network, we need to model GRID applications as represented on figure 1 and described below. The model is composed of five layers. The first one is a representation of a network and, in each layer, we have special mechanisms and protocols like a routing protocol, messages passing... At the top, layers are assigned to GRID management and its applications.

Layer 1 : network. A network is modeled by a graph $G_N = (V, E)$, where V is the set of nodes and E the set of edges. Each node represents a computer (we can use computers, processors or resources indifferently) of the network and each edge represents a physical link: $\forall n_1, n_2 \in V^2, (n_1, n_2) \in E$ if n_1 and n_2 are connected physically. Each node is associated with an unique identifier and communicates with its neighborhood by passing messages through communication links. We consider the network to be asynchronous. Each node owns a set $Neigh_i$ which contains its neighborhood and is automatically maintained.

Layer 2 : routing. In this layer, a routing protocol computes local routing table in each node. For example, in [9], the author shows how to build them with a random walk. But we can use classical algorithms like *distance vectors* or *link states*. We define the sets as called $Reach_i$; they contains all the network nodes that node i can reach. $Reich_i$ are automatically maintained. So, we build a new graph G_R from these sets. With restricted access rights by firewall, for example, G_R has directed links and if there are few firewalls, we can suppose that this graph is dense. We obtain $G_R = (V, E')$ with $E' = \{n_1, n_2 \in V^2, (n_1, n_2) \in E'$ if a path exists from n_1 to $n_2\}$.

Layer 3 : communications. A communication protocol works on G_T in order to exchange messages between nodes that have a route between them. This protocol

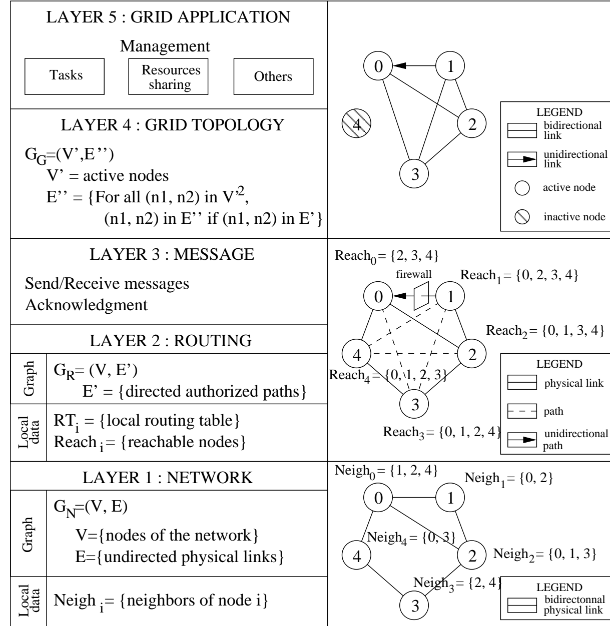

Fig. 1. On the left, figure shows layers of our model and on the right, a network and its representation through each layers.

supplies functions *send* and *receive* with errors and acknowledgments management: if a message is not transmitted to the destination node, the sender of the message is informed. This layer also ensures the correctness of received data and corruptions can be detected. The network is asynchronous, but we suppose we have an upper bound on communication delay.

Layer 4 : GRID topology. In this layer, we distinguish two node states. When a node enters the GRID its state is called *active*. Conversely, a node of the network that does not collaborate with the GRID is called *inactive*. Within layer 3, this node keeps relaying messages. We construct a graph $G_G = (V', E'')$ with $V' = \{n \in V/n \text{ is active}\}$. $V \backslash V'$ is the inactive nodes set. The set of links of G_G is $E'' = \{\forall n_1, n_2 \in V'^2, (n_1, n_2) \in E'' \text{ if } (n_1, n_2) \in E\}$. We have the following properties: $\forall n_1, n_2 \in V'^2, \exists$ path between n_1 and n_2 and $G_G \subseteq G_R$. The $Reach_i$ sets contain active and inactive neighbors of nodes.

Layer 5 : GRID application. Applications that work on the GRID must add several kinds of management: task management, resources sharing... In this paper, we focus on computational GRID. So we only describe tasks management that has to distribute tasks into the GRID and to collect results.

We use the solution introduced by CONFIIT. Each task of a problem is

tagged by a state: *uncomputed*, *computed* or *in progress*. Each node owns an array that contains the whole set of tasks with its state. A random walk circulates into the GRID containing a copy of the sender local array. It updates local arrays of each GRID node. When a node wants to compute a task, it chooses at random one of *uncomputed* tasks in its local array and tags it with the *in progress* state. At the end of the computation, the task is tagged *computed* and the node chooses another one. When all tasks are computed, the problem is finished.

In order to avoid uncomputed tasks and to accelerate the global computation, when a node has not any uncomputed task, it chooses randomly one of the *in progress* tasks. For example, we can have a node that chooses a task and crashes before terminating it. This task is considered as a *replicated task*.

3.2 Fault impacts

In each layer of our model, we can have internal faults or impacted ones from lower layers.

Layer 1. We can only have definitive breakdowns in this layer: a node can fall down or a link can be disconnected. If a link $(i, j) \in E$ is disconnected, nodes i and j update their neighborhood. A node crash is a disconnection of all its links: $\forall j \in Neigh_i, Neigh_j \leftarrow Neigh_j \backslash \{i\}$.

Layer 2-3. When a fault is impacted from layer 1, the routing protocol has to compute new paths. If it succeeds, local routing tables are updated and higher layers do not detect any fault. Otherwise, we can have path loss and some nodes can still be unreachable. They are supposed to be disconnected.

Layer 4. This layer ensures the GRID topology management. If some nodes are disconnected, there are deleted from the GRID. We also have to manage deactivation of GRID nodes. If one of them wants to go out of the GRID (voluntarily or not), the protocol has to update the topology. In the same way, if we use a token for the protocol, it can be lost when a node crashes. So we need a recovery mechanism.

Notice that when a node crashes, the task that it is computing is deleted. It must be computed by another node.

Figure 2 shows the fault management scheme.

4 Topology management with a random walk

4.1 Introduction

The GRID topology (layer 4) is updated by a token. The state of the GRID is an adjacency matrix. Each node i owns a local matrix M_i defined by:

$$\forall i, j \in V'^2, M_i(i, j) = \begin{cases} 1 \text{ if } j \in Reach_i \\ 0 \text{ if } j \notin Reach_i \end{cases}$$

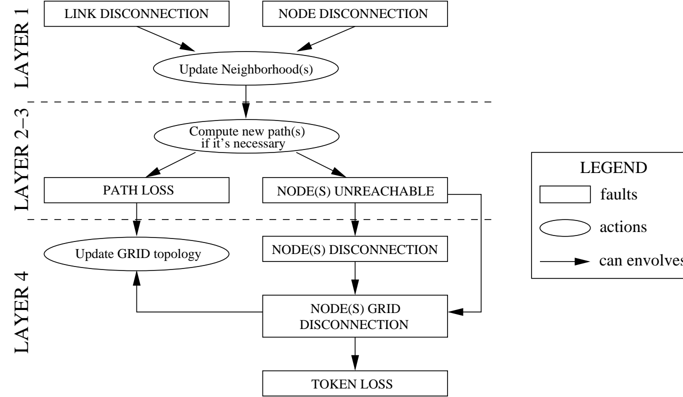

Fig. 2. Impacts of faults through layers of our model

The topology token contains a copy of the sender matrix M_T. It is used to update the local matrices of other nodes.

To manage tasks (layer 5), we use the solution introduced in the CONFIIT middleware described in section 3.1. We need a token containing a state tasks array A_T which is a copy of the local array of the sender node. State tasks are updated in each node.

The two tokens circulate at random in the GRID. We choose to merge them into a single token to reduce the number of messages.

4.2 Algorithm

When a node receives the token (M_T, A_T), it updates its local matrix and its local array. When it is finished, the token is sent with M_i and A_i. To update the local matrix, we can distinguish the following cases:

- $\exists n \in M_T / n \notin M_i$: a new node n has integrated the GRID. n must be inserted into matrix M_i. We use the *Add* function which adds a new row and a new column and updates neighbors.
- $\exists n \in M_i / n \notin M_T$: the token does not contain node n. Only node i knows this node. It is a new one and its entry point is node i.
- $\exists n \in M_T$ and $n \in M_i$: we update neighbors for node n into M_i with the *Update* function.

$State_i(t)$ represents the state of a task t in the array of node i. It values either uncomputed (0), or in progress (1) or computed (2). The function called *UpdateTasks* update the local array A_i by keeping the better task progression to update each task state. Algorithm 1 shows the reception of a token (M_T, A_T) by a node.

Algorithm 1 *Token reception* from a node j in node i

```
1: // Add new nodes and update M_i
2: for all node n_T ∈ M_T do
3:    if n_T ∈ M_i then
4:       Update M_i
5:    else
6:       Add n_T into M_i
7:    end if
8: end for
9: UpdateTasks
10: // Send the token
11: Send TOKEN to a random neighbor with M_T ← M_i and A_T ← A_i
```

4.3 Connection to the GRID

When a node wants to enter the GRID, it has to contact one of the GRID nodes called an *entry point*. We need a bidirectional link between the new node and the entry point. The new node sends a *connection request* message which contains its neighborhood. The entry point answers by sending its adjacency matrix and its local tasks states array. In case of success, the two nodes update their local matrix. The new node is integrated into the GRID and can compute a task. With its local knowledge, it becomes a possible entry point. The entry point also has to verify if there are only two nodes in the GRID. In this case, it has to create the GRID token and to send it to the new node.

4.4 Fault management

On layer 4, we have to solve faults impacted from lower levels of our model. As we have shown in section 3.2, we need to manage nodes and links disconnection. A node detects it with the automatic updating of its set *Reach* by layer 2. We also have to manage the deactivation of a node and the token loss.

Link or node disconnection. When a node detects a neighbor disconnection, it cannot know if it is a link or a node disconnection. So, it only updates its adjacency matrix. With the latter, the node can build a spanning tree rooted on it. If it detects that some nodes are not included in this tree, these nodes are unreachable and can be deleted from the matrix: they are considered as inactive nodes.

GRID node deactivation. A GRID deactivation can be voluntary if a user wants to stop its collaboration. In this case, it can send messages to its neighbors to inform them about it. But if the application suddenly crashes, it cannot send any messages. It becomes an inactive node but its disconnection is not yet detected by its neighbors: it is still in the network and neighborhoods, but it cannot response to GRID messages. When one of its neighbors receives the token and tries to send it, an error occurs. So this neighbor can update its local matrix. To ease the management and to avoid any message creation, we do not distinguish these

two cases. For a voluntary disconnection, nodes have not to send any messages.

Token recovery. According to layer 3, the data into the token can not be corrupted. In the same way, the token can not be lost: when a node sends a message, it has an acknowledgment. But when a node crashes after reception of the token and before sending it to one of its neighbors, the token is lost. So each node owns a timeout T_i that allows the regeneration of a token when it ends. In this case, node i updates its local matrix with its neighborhood knowledge: M_i only contains its neighbors. For each one, the node tests its reachability in order to delete inactive ones. Then, the node chooses a random neighbor and sends the token. The initial value of the timeout depends on the GRID size. If the value is too low, many tokens will be created and, on the other hand, if the value is too high, a token loss induces a reaction time before regeneration.

After the crash of a node or a link, it is possible for the GRID to contain more than one token: the timeout can ends before the coming of the token. We have to add a new mechanism to delete the additional ones. In [13], the authors choose to mark the token with a sequence number *seq* and the identity of node *id* that creates it. Each node keeps a memory of the sequence and identity of the latest token. When a node enters the GRID, its sequence number and *id* equal 0. When a node generates a new one, it increases this sequence number. Then at reception of a token with sequence *newSeq* and identity *newId*, we have the three following cases to consider:

- $seq > newSeq$: the token is an old one so it can be destroyed.
- $seq < newSeq$: the token is more recent than the latest one, so *seq* and *id* can be updated.
- $seq = newSeq$: it is a normal case according to the value of *id* and *newId*:
  - $id = newId$: normal case.
  - $id < newId$: the token is more recent than the latest one, so *seq* and *id* can be updated.
  - $id > newId$: token is an old one so it can be destroyed.

4.5 Full algorithm

When a node receives the token, it has to test the reachability of its neighbors. For each one, it uses the function *reachable(n)*, that returns *true* if node n is still reachable: $n \in Reach_i$ and $n \in G_G$. It can update its local matrix. Then, from the matrix, it can build the spanning tree rooted on it in order to discover detached nodes and to delete them. Algorithm 2 shows the full process to remove all the unreachable nodes of the GRID.

Algorithm 2 Removing unreachable nodes

```
1: // Delete inactive nodes
2: for all node n_i ∈ M_i do
3:    if n_i ∉ M_T then
4:       if reachable(n_T) == false then
5:          Delete n_i into M_i
6:       end if
7:    end if
8: end for
9: // Search unreachable nodes
10: T_i = tree rooted in i
11: Delete all nodes n ∈ M_i / n ∉ T_i
```

In algorithm 3, we describe the behavior of a node which receives the token. First, we have to test the validity of the token according to its sequence. Then, the node updates its local matrix. When it has to add a new node included in its neighborhood, it must test if this node is already active.

Algorithm 3 *TOKEN* reception from node j at node i

```
1: if sequence of TOKEN is not valid then
2:    Destroy TOKEN
3: else
4:    // Add new nodes and update M_i
5:    for all node n_T ∈ M_T do
6:       if ∃n_T ∈ M_i then
7:          Update M_i
8:       else
9:          if n_T ∈ Reach_i then
10:            if reachable(n_T) == true then
11:               Add n_T into M_i
12:            end if
13:         else
14:            Add n_T into M_i
15:         end if
16:      end if
17:   end for
18:   Removing unreachable nodes
19:   Update tasks array
20:   // Send the token
21:   Send TOKEN to a random neighbor with M_T ← M_i and A_T ← A_i
22: end if
```

5 Simulations

We have developed a C++ program to perform simulations based on our random walk solution. We use the decomposition of a Langford problem in irregular and independent 1366 tasks. Length tasks are given thanks to a computation on the CONFIIT middleware. Thus we obtain very irregular lengths (2,000 ms to 18,432 ms). A sequential computation takes $T_s = 14,913,112\ ms$.

We launch simulations on several random networks so as to obtain an average time to finish the whole computation T_t and an average number of the replicated

tasks. In order to compare the distributed solution with the sequential one, we compute an efficiency factor e that depends on the size of the network (n is the number of nodes) and is given by the following formula:

$$e = \frac{T_t * n}{T_s}$$

If a network contains n nodes, an optimal distributed solution must spend n less time to finish computation and has an efficiency factor equal to 1.

We launch a number of simulations on random networks with 10 to 100 nodes. For each series, we fix the network and we obtain the results on figure 3.

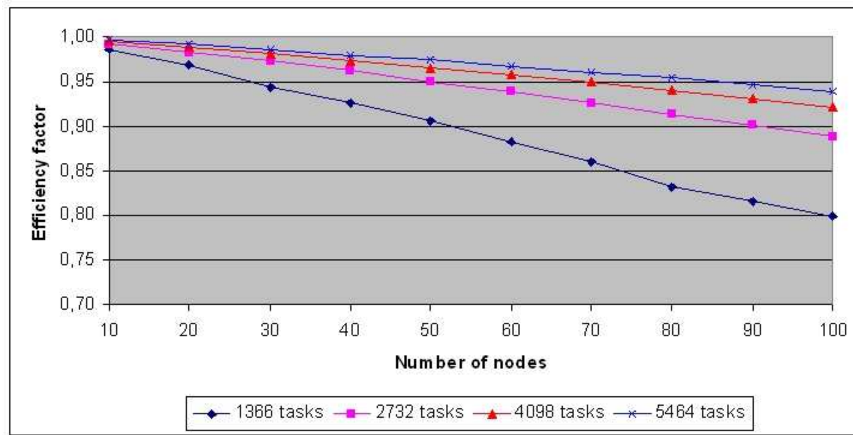

Fig. 3. Efficiency factor in function of network size

In order to show the breakdowns tolerance, we add voluntary and regular node disconnections during a fixed length. After a breakdown, the node wakes up and sends a connection request to one of its neighbors. We launch the simulations on a fixed random network with 100 nodes. We increase the frequency of breakdown and we obtain the results on figure 4.

The more the network grows, the longer the time to cover the graph. It induces a reaction time to inform other nodes and increases the likelihood of a node choosing the same task as an other node. It produces more replicated tasks and the efficiency factor decreases. For these simulations, we have 1366 tasks to compute for 100 nodes: only 14 per node.

As far as breakdowns are concerned, we notice a number of replicated tasks triggered by the tasks management. A node starts a task. It receives the token that informs the other nodes that the task is in progress. If this node crashes, the task is tagged as *in progress* but it cannot be computed. With a larger number of breakdowns, after a given time, local task arrays contain only *in progress* tasks.

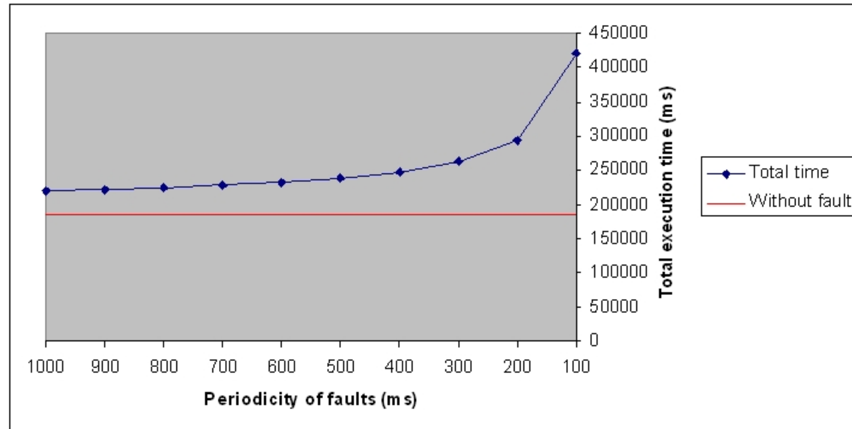

Fig. 4. Impacts of periodic breakdowns

When a GRID node selects one of them, it supposes that it is a replicated one but in fact, this task is computed for the first time.

6 Optimizing token content

For an adaption to a large network, we have to consider a number of nodes. The size of the matrix becomes too large and overloads the network. We can reduce local matrix M_i and token matrix M_T by keeping only the useful data: the directed spanning tree. To build it, we introduce the *predecessor* notion: when node i receives the token from node j, j becomes its predecessor. The tree is represented in node i as an array G_i which contains the predecessor of each GRID node. When a node crashes, we allow only its predecessor to delete it from the GRID. For example, figure 5 shows a GRID with a node crash. Node 3 receives the token and detects that node 4 is disconnected. We build a spanning tree from array G_3. It shows that node 3 is the predecessor of node 4, so it can destroy it and its subtree from the GRID nodes set.

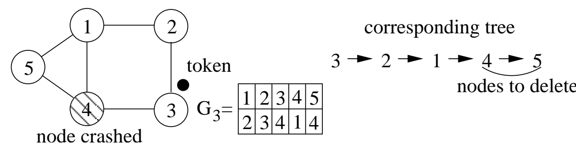

Fig. 5. Example of a node crash with predecessor solution

This solution reduces local data and length of the token but some problems occur. If the token produces a cycle, we can not build a spanning tree.

7 Conclusion

We show that the topology management using a random walk is an efficient solution as shown by simulations. It allows many topological changes without GRID falls and without the management of a virtual structure. But we notice that the growth of the network implies a decreasing efficiency. The size of the token and data into each node must be reduced to keep a good performance level. Some solutions can be presented. For a larger network, we are planning to use multiple random walks and clusterings in order to reduce replicated tasks by increasing the update of local task arrays.

Acknowledgments

This work was partly supported by "Romeo"[2], the high performance computing center of the University of Reims Champagne-Ardenne, the "Centre Informatique National de l'Enseignement Superieur"[3] (CINES), France and by the project ACI GRID-ARGE funded by the French ministry of research.

References

1. W. Allcock, A. Chervenak, I. Foster, L. Pearlman, V. Welch, and M. Wilde. Globus toolkit support for distributed data-intensive science. In I. Press, editor, *Computing in High Energy Physics (CHEP'01)*, September 2001.
2. D. P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.
3. D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, November 2002.
4. A. Bui, M. Bui, and D. Sohier. Randomly Distributed Tasks in Bounded Time. In T. Böhme, G. Heyer, and H. Unger, editors, *IICS*, volume 2877 of *Lecture Notes in Computer Science*, pages 36–47. Springer-Verlag, December 2003.
5. E. Caron, F. Desprez, F. Lombard, J.-M. Nicod, M. Quinson, and F. Suter. A Scalable Approach to Network Enabled Servers. In B. Monien and R. Feldmann, editors, *Proceedings of the 8th International EuroPar Conference*, volume 2400 of *Lecture Notes in Computer Science*, pages 907–910, Paderborn, Germany, August 2002. Springer-Verlag.
6. G. Fedak, C. Germain, V. Neri, and F. Cappello. XtremWeb: A Generic Global Computing System. In *IEEE/ACM - CCGRID'2001 Special Session Global Computing on Personal Devices*. IEEE Press, May 2001.

[2] http://www.univ-reims.fr/Calculateur
[3] http://www.cines.fr

7. U. Feige. A tight lower bound on the cover time for random walks on graphs. *Random Structures & Algorithms*, 6(4):433–438, 1995.
8. U. Feige. A tight upper bound on the cover time for random walks on graphs. *Random Structures & Algorithms*, 6(1):51–54, 1995.
9. O. Flauzac. Random Circulating Word Information Management for Tree Construction and a Shortest Path Routing Tables Computation. In R. G. Cardenas, editor, *OPODIS*, Studia Informatica Universalis, pages 17–32. Suger, Saint-Denis, rue Catulienne, France, 2001.
10. O. Flauzac, M. Krajecki, and J. Fugère. CONFIIT : a middleware for peer to peer computing. In C. T. M. Gravilova and P. L'Ecuyer, editors, *The 2003 International Conference on Computational Science and its Applications (ICCSA 2003)*, volume 2669 (III) of Lecture Notes in Computer Science, pages 69–78, Montréal, Québec, June 2003. Springer-Verlag.
11. I. Foster and C. Kesselman. Globus : a metacomputing infrastructure toolkit. In I. Press, editor, *Supercomputer Applications*, volume 11 (2), pages 115–128, 1997.
12. I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. MORGAN-KAUFMANN, September 1999.
13. M. Krajecki, O. Flauzac, and P.-P. Mérel. Focus on the communication scheme in the middleware confiit using xml-rpc. In *18th International Workshop on Java for Parallel Distributed Computing (IW-JPDC'04)*, volume 6, page 160b, Santa Fe, New Mexico, April 2004. IEEE Computer Society.
14. S. Li. JXTA 2 : A high-performance, massively scalable p2p network. Technical report, IBM developerWorks, November 2003.
15. L. Lovász. Random walks on graphs : A Survey. In T. S. ed., D. Miklos, and V. T. Sos, editors, *Combinatorics : Paul Erdos is Eighty*, volume 2, pages 353–398. Janos Bolyai Mathematical Society, 1993.
16. Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 84–95. ACM Press, 2002.