Distributed Spanning Tree Algorithms for Large Scale Traversals

Sylvain Dahan
Laboratoire d'Informatique de l'Université de Franche-Comté
16, Route de Gray - 25030 Besançon cedex - France
Email: dahan@lifc.univ-fcomte.fr

***Abstract*— The Distributed Spanning Tree (DST) is an overlay structure designed to be scalable. It supports the growth from small scale to large scale. The DST is a tree without bottlenecks which automatically balances the load between the nodes.**

This sound paradoxale, but the DST breaks the common assumption that a tree is build of leafs and intermediate nodes. In a DST every nodes are equal. The nodes are put together into small cliques. Then, the cliques are put together into small cliques of higher level recursively. The cliques are represented in each node by a routing table. The memory space complexity of the routing tables is $\mathcal{O}(\log(n))$ for a n nodes DST.

A theorical description of the DST was already published but it does not provide enough information about the capacities of the DST. The purpose of this article is to give a practical view of what can be done with a DST. This document can be considered as a DST traversal catalog. It also outlines some characteristics and shares the lessons learned from our implementation errors.

Keywords: Unstructured overlay network, Grid, scalability, traversal algorithms, fairness, data structure.

I. Introduction

GRID applications federate huge set of resources. Those resources can be files, databases or processing units. They need communication and discovery mechanisms. Without them, the resources are unable to cooperate and the system is unusable. Simple cluster mechanisms like cliques or centralized schedulers do not support large scale systems and can not be used by GRID applications. Thus, peer-to-peer mechanisms become an interesting scalability model for the GRID.

Peer-to-peer applications are mostly build on top of overlay networks. The surveys [1] and [2] are good introductions about them. Some overlays are used to build distributed hash tables. The distributed hash tables are well to implement indexes and rendez-vous points. But, those are not the only mechanisms that need overlays. The tree is also an overlay network used by several distributed algorithms. However, trees have bottlenecks and are not scalable overlays. Thus, peer-to-peer applications prefer random graphs to trees.

Usually, to remove a bottleneck, distributed algorithms share the load between several components. The distributed spanning tree (DST) is a theoretical structure which is able to distribute the load of the root and the intermediate nodes of a tree toward its leafs. This is possible because, the intermediate nodes do not have a physical existence in a DST. However, the DST is organized into a hierarchy identical to the tree by the following way. The leafs are put together into small groups. Those groups are the DST intermediate nodes. Then, recursively, the groups are put together into small groups of higher level creating another level of intermediate nodes. This way, the DST is able to fairly share the load of the structure between all the participants and to keep all the advantages of the tree structures.

The theorical aspects and related works of the distributed spanning tree are described in [3]. The DST is, in the same time, a hierarchical and a completely fair structure. Thus, the DST is disturbing at the first sight and theorical descriptions are not enough to understand the whole capacities of the DST. Despite that the structure and its algorithms are really simple, it was difficult to use the DST at its full potential. We made several mistakes on how-to operate a DST. Sometime seeing it as a simple tree and sometime forgetting that the DST is also a tree.

The aim of the present article is to share our experiences by explaining the various ways to operate a DST. The correctness proofs of the presented algorithms are straightforward and are not included in this document leaving more space for the explanations. This article can be viewed as a list of traversal algorithms corresponding to different use cases. Those use cases present progressively the different aspects of the DST and are illustrated with an example.

This paper starts by a practical description of the distributed spanning tree data structure. Then, the section III describes the main concepts that should be used to operate a DST at its optimum. The following sections (IV–X) introduce different traversal algorithms running on top of the DST. Finally, some hidden faces of the DST are discussed before the conclusion.

II. The Data Structure

The distributed spanning tree structure can be studied at 3 different levels. This section describes the 2 higher levels: the logical level and the interconnection level.

A. Logical Level

The example drawn in Fig. 1 is used all along this document. This DST has 21 nodes. Nodes are represented with plain line rectangles. Each node has a unique and permanent address which is a, b, ..., or u. At the logical level, nodes are called subject. Subjects have a virtual name which is here a 3 digits number. The nodes are organized into a hierarchy of groups.

Groups are represented with dotted line rectangles. Groups also have a virtual name.

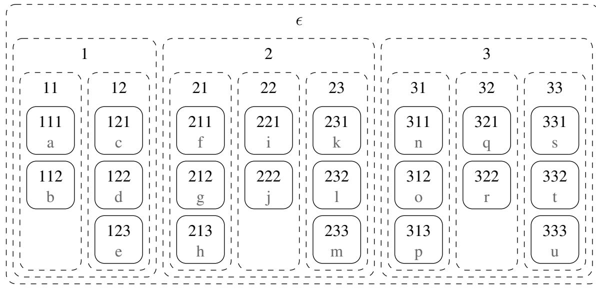


Fig. 1. Logical level of the example

Depending of the level, an element can be a subject or a group. The elements virtual name depends on their level and their position. Here, a virtual name is always a x digits number. x is equal to the number of levels minus the group level number. Subjects have the level 0. So, in the example subjects have 3 digits number virtual name, groups of level 1 have 2 digits number virtual name and the group of level 3 has the 0 digit number virtual name, ϵ.

The elements are indexed in their group. The last digit of an element virtual name is the index of the element. The other digits are the virtual name of the group that contains the element. All the elements of a group must have a unique index. Here, the index is an integer that is bounded by 1 and the number of elements in the group. The Fig. 1 and the Fig. 2 respect this naming convention.

The logical level of the DST structure is defined by the following properties:

1) A group has between a and b elements. An element can be a group or a subject depending of the group level.
2) The top level group can have less than a elements.
3) There is only one top level group.
4) The subjects are put together to form groups of level 1.
5) If i is not the top level, the groups of level i are put together to form groups of level $i + 1$.
6) Every element of a group must know all the other elements of its group.

The example has 3 levels, so its height is 3. The example sets a at 2 and b at 3. Those values have been chosen to draw aesthetics figures. But in real applications, having $2 \leq a \leq \frac{b}{2}$ is strongly recommended to simplify the algorithms that insert or remove a node.

The hierarchical structure of the DST is the same as a tree where groups act as fathers and elements act as children. The Fig. 2 clearly shows the example as a tree by displaying it with another format.

The DST is fully distributed because the load of a group is fairly shared between its elements. This is true because the groups delegate their works to their elements. For example, when ϵ needs to send a message to 1, 2 and 3, this is one of these 3 elements which sends the message to the 2 others. This is why all the elements of a group must know each others. Or more formally, the elements of a group must form a clique. The DST avoids to use always the same element to balance the load between the elements of the groups.

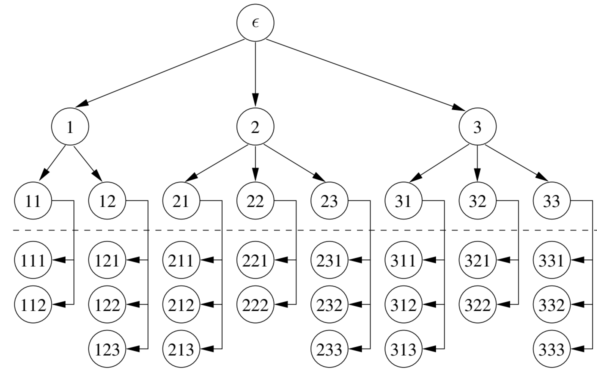


Fig. 2. Tree representation of a logical level

B. Interconnection Level

The logical level explained that groups delegate their works to their elements. This is a recursive process and finally, it is the subjects that do all the works. In the interconnection level subjects and groups are abstract elements. Nodes are the real entities which act as subjects and groups.

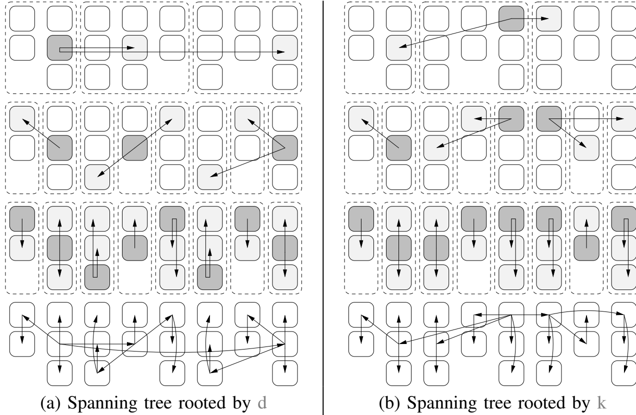
The DST is linked by routing tables shared by all the nodes. The table I shows the routing table of the node d. The routing table has one row by DST level. So, the example routing tables have 3 rows. Each row stores the addresses of the elements of a group. But it is not possible to have the address of a whole group, because groups are abstract entities in the interconnection level. Instead, it stores the address of a node for each element of the group. For example, the node d, is member of 1. So, it uses the node d to represent 1, the node j to represent 2 and the node t to represent 3. The node d is also member of 12. So, it needs to knows an 11's node to uses it as 11's representative and a 12's node to uses it as 12's representative. As shown in table I, each time a node needs a representative for one of its group, it uses itself as representative. This is to avoid unnecessary links. So, the node d uses itself as representative of the groups ϵ, 1 and 12. Every node has a routing table build on the same model. The different cliques can exist because each node has a link toward a node for every element of its group for each level.

TABLE I
THE NODE D'S ROUTING TABLE

position		representatives		
lvl	idx	1	2	3
3	1	*d*	j	t
2	2	a	*d*	–
1	2	c	*d*	e

The table I also displays the index of the node d for every level. By reading the indexes from top to down, we get {ϵ, 1, 12, 122} which are the virtual names of the different elements that contain the node d.

The Fig. 3 shows how to build spanning trees with a DST. Two trees are build in Fig. 3. One rooted by the node d and another one routed by the node k. In the first step, the links of the top level group are used to contact a node of each element of the group of level 3. Then, in the second step, each contacted node uses its links of level 2. Now each group of level 1 has been contacted. In the third and last step, the links of level 1 are used and the whole set of nodes are contacted. The last pictures show the resulting spanning trees. This way, each node can have its own mapping of the DST which is distinct of the others.



(a) Spanning tree rooted by d (b) Spanning tree rooted by k

Fig. 3. Spanning trees build from a DST

Two hypothesis are made about load balancing. Firstly, the DST supposes that each node has the same probability to be an initiator and nodes use the spanning tree rooted by themselves. This is important because the routing tables of the DST are static. They are easy to update when a node joins or leaves the structure or when a bottleneck is detected. But it is impossible to get a new representative each time that the structure is traversed. If only one or two nodes are used as root, the DST behaves like all the other trees and lost its main interest.

The second hypothesis is that the algorithms that insert or remove a node are careful about the DST balance. By balance, we mean that each group should mostly has the same number of nodes and every node should be the representative of its groups the same number of times. If a node is used as representative by a lot of nodes when other nodes are not used as representative, the load is not equitably shared. However, this hypothesis is not inviolable. Some applications do not need a strong balance, others break it to put more load on powerful computers.

III. Advantages of the Distributed Spanning Tree

The data structure of the distributed spanning tree has been presented above. This section describes the main advantages of the DST. To be able to make an optimal use of the structure, the associated algorithms should take care about the concepts explained below.

As a tree, to contact n nodes, only n messages are needed. And $2n$ messages are needed to query n nodes. This can be always true and all the presented algorithms use at most $2i$ messages to query i nodes.

The structure has its own hierarchical organization. The traversal algorithms should use this organization to know which nodes have been already visited or not. Then, it becomes useless to stamp the visited nodes or to generate a list of visited nodes. At most, $\mathcal{O}(\log(n))$ information are needed to store the state of a traversal. This correspond to one piece of information for each level of the DST.

Each node of a DST chooses different nodes as representatives of the elements of each group. Usually, each node represents a group for the same number of times that the other nodes of the group. So, if each node has the same probability to initiate a traversal, each node has the same probability to be used as representative of a group. We call it the equiprobability rule.

If all the nodes of a DST are contacted and if each node has the same probability to initiate a multicast, then the structure is able to fairly share the load between the nodes. But some mechanisms need only partial traversals and contacting the whole set of nodes is only done in the worst cases. Those algorithms should care about fairness.

If an in-order traversal is done, the nodes are always contacted in the same order. If only partial traversals are done, the first node is visited every time and the last one can be never visited. This is against the fairness concept of the DST. Instead each node must be careful about its traversal order which should be harmonized with the traversal order of the other nodes. We call it the fairness rule.

Finally, it is possible to start a traversal locally. It is usually better for a node to visit its own lower group before starting to visit distant groups. This is useful to build a locality approach that are needed by some heuristics. As a fully distributed structure, the DST supports well this kind of optimization without adding cost.

IV. A Parallel Traversal Algorithm

The first presented traversal is a parallel traversal algorithm. This is the simplest one and the key ideas have already been explained in section II. The aim of a multicast is to send a message to every node as fast as possible.

The parallel traversal algorithm sends recursively the message $\langle i \rangle$. Each node represents several elements. In the example, a node represents itself, a group of level 1, a group of level 2 and the group ϵ of level 3. i indicates the level that receives the message. For example, if the node d receives the message $\langle 2 \rangle$, it knows that the addressee is the group 12, because the group 12 is its group of level 2.

When a node receives the message $\langle i \rangle$ with $i > 0$, it sends the message $\langle i-1 \rangle$ to every element of its group of level i. $i = 0$ means that the node has received the message as a leaf and it does not forward it. If h is the number of levels of a DST, a node send $\langle h \rangle$ to itself to initiate a multicast. Then recursively, the message goes down through each level and is received by every node. The Fig. 4 shows step by step a parallel traversal initiated by the node d.

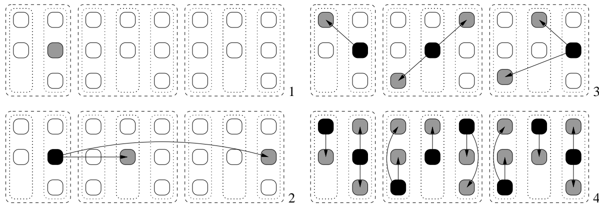


Fig. 4. Parallel traversal initiated by d

V. A Breadth First Search Algorithm

The breadth first search algorithm is used by several discovery mechanisms like the Gnutella protocol [4] when flow control or ultrapeers are not available.

The breadth first search is used to query the nodes step by step. In the first step, the root is queried. After, the children of the root are queried. Then, it is the turn of the grandchildren. And so on. The characteristic of this algorithm is that at each step the number of queried nodes increases exponentially.

With a DST and some distributed algorithms, it is not possible to contact directly the grandchildren of the root, because there is no centralized knowledge of the structure. Each node knows only its neighbors. In a tree, the neighbors of a node are its father and its children. Thus, to query its grandchild, a root node need to pass through its children. The Fig. 5 is an example of this breadth first search traversal.

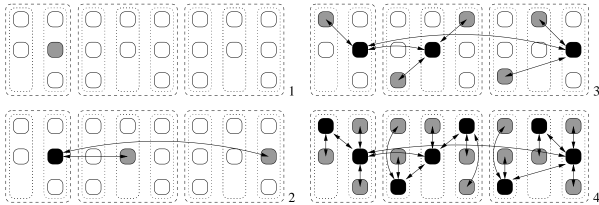


Fig. 5. A naive breadth first search initiated by d

We discover that this DST traversal is not optimal. The need to pass each time through its descendants to go a step ahead is an overhead. The DST can avoid this overhead. So, mapping a classical tree breadth first search algorithm directly on a DST is not the better solution.

The main interest of the breadth first search is the number of queried nodes for each step. Most of the time, the number of queried nodes is significant and the order of the traversal is not an problem. In this case, by implementing the locality concept (§ III) only $2i$ messages are needed to contact i nodes.

Instead of starting by the root and going down progressively, the DST breadth first search algorithm starts from a leaf and goes up. The initiator queries itself as a leaf. After, it queries the other leafs of its group of level 1. Then it queries all the nodes of its group of level 2 but its nodes of level 1. And so on. By queried recursively all the nodes of its group of level $i+1$ without contacting the nodes of its group of level i, it is impossible to query a node twice.

The Fig. 6 shows a DST breadth first search traversal. Firstly, the node d query itself as a leaf. Then it queries the other nodes of the group 12 which is its group of level 1. It can pursue by querying the nodes of the group 11 which is the other element of its group of level 2. To finish, it queries the nodes of the group 2 and 3 which are the other elements of its group of level 3, ϵ.

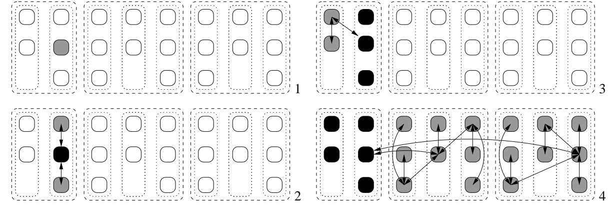


Fig. 6. An optimal breadth first search initiated by d

By querying successively groups of higher level, this traversal queries a number of nodes which grows exponentially. This is true, because the number of nodes inside a group grows in average exponentially with the level of the group. The number of elements of a group of level i is always bounded by a^i and b^i.

VI. A Limited Depth Parallel Algorithm

The main problem with the bread first search traversal is that the number of queried nodes grows exponentially. This can cause serious scalability drawback as explained in [5]. To resolve this issue, a limit is implemented. A maximal depth is set to avoid that to many nodes are queried in parallel for a unique request.

But another problem comes. What appends when the result of a traversal is not successful. Some mechanisms return a failure, other try a new traversal. Doing the same traversal again and again does not always resolve this problem. Some mechanisms try to run a new traversal which queries new nodes. This was an approach used by Mutella, a Gnutella servants. Despite that this new feature seem to come from a bug of Mutella [6], the idea is interesting. Its connections does not last more than few seconds. Thus, it always jumps from place to place in the Gnutella network. If the servant waits few minutes between two limited depth parallel traversals, it is certain that new nodes are queried.

This method can also be used on a DST but in a most efficient way. Querying every element of a group of level i is like a parallel traversal of depth i. The parallel traversal algorithm (§ IV) can be easily used.

To query new nodes, the algorithm queries a group which has not been already queried. This is similar to a sequential traversal, despite that the groups of level i are considered as leafs. Sequential traversals are explained below. The Fig. 7 is an example of this algorithm with a depth limited at 1. The centralized traversal (§ VIII) is used in this example to travers the groups one by one.

VII. A Sequential Algorithm

The previous sections described parallel traversals. The following sections focus on strictly sequential algorithms.

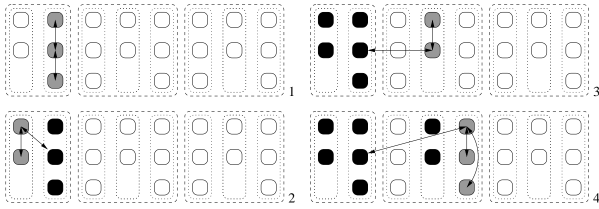


Fig. 7. Limited depth parallel traversal initiated by d

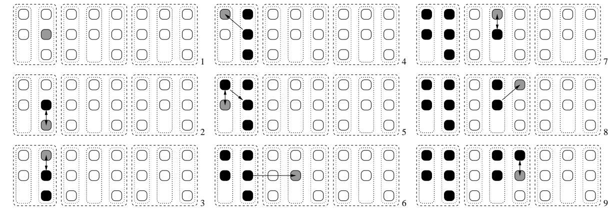


Fig. 8. Sequential traversal initiated by d

Finding a good sequential traversals algorithm was not a trivial task. In our applications, we always use the traversal algorithm to do partial traversals and prefer parallel ones when every node must be visited. But the issue presented here should also be interesting for complete sequential traversals. Our first trials were complete failures because some nodes were overloaded when other were underloaded. At that moment, we were not aware about the fairness rule (§ III).

The issue did not come from the DST, but came from our bad usage of it. We thought that the locality concept was enough. We thought that visiting the lowest groups and visiting successively the upper groups will balance the load between the nodes. This approach improves greatly the load balancing but it was not enough.

When the elements of a group were visited, we always start by the local element. The local element is the initiator of the traversal, or the representative element that received a query from its father group. Then the remaining elements of the group were visited one by one by following their index number. Thus the elements inside a group were visited approximately in the same order. Then, the elements with a low index got more load than the others.

We found several methods to resolves this problems like random path, consensus between the elements or overload detection. But some of them are very difficult to implement and the cure was worst than the disease. The random path was the only usable algorithm before we found a better one.

The bottlenecks did not come from the fact that the elements were visited in following the index numbers. It came from the fact that the first ones were visited before the others. Our best heuristic is to follow the elements index number starting by the representative element. From the equiprobability rule (§ III), which said that every node has the same probability to initiate a traversal and each node has the same probability to be used as a representative of one of its group, all the elements of a group as the same probability to be a representative element. Visiting the elements by following their index number, starting by the current representative element, gives to every elements of the group the same probability to be visited.

The Fig. 8 is an example of this sequential traversal. The index number of the elements can be find in Fig. 1. The last digit of their virtual name is the index number (see § II-A) used by the sequential algorithm.

VIII. A Centralized Algorithm

The use of a centralized algorithm with a DST is not an antinomy. One of the interesting result of the Andrew and Coda File System about scalability is: Whenever there is a choice between performing an operation on a client and performing it on a server, it is preferable to pick the client [7].

This algorithm put all the load on the initiators of the traversals. This is possible and the overhead is small. A node can not know the whole structure because there are too much nodes in a DST. But a complete DST traversal can be done because each node has its own routing table that allows it to send messages to the elements of its groups. The size of those routing tables are small and can be sent to every node which requests it. But a node can not store all these routing tables at once.

With the previous algorithm (§ VII), $2i$ messages are needed to contact i nodes. One for the request and one for the answer. If a node queries directly i nodes, $2i$ messages are also needed. Furthermore, those two methods take the same time to send all their messages.

The traversals are done with the information stored inside the routing table of each node. If a node is able to get those information, it can do a traversal by itself. But the node must gather this information in parallel of the traversal, because it can not store the routing tables of the whole DST. This is possible, if each node that is queried returns, with its response, its routing table. The overhead is small because the size of a routing table is small and these information are encapsulated in the response. If n is the number of nodes of a DST, then each node stores only $\lceil \mathcal{O}(\log(n)) \rceil$ information.

As all the other centralized tree traversals, the initiator node uses a stack to store the state of the traversal. Each element of the stack is the routing table of an intermediate node. In our case, the intermediate nodes are nodes that were used as representative of a group.

The Fig. 9 shows a centralized traversal. The traversal order is exactly the same as the Fig. 8. This is normal, because the same routing tables are used in the two cases.

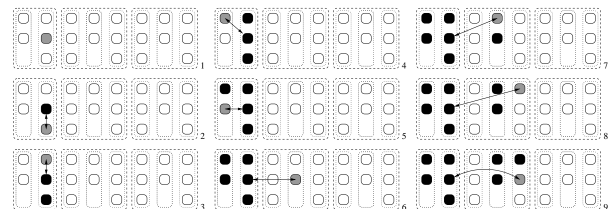


Fig. 9. Centralized traversal initiated by d

To resume, by returning the visited node routing table to

the initiator node, the initiator is able to do a traversal and to query every node by itself. This selfish view can be interesting. If the system helps everybody and every node works for the others, it is possible that few nodes launch lot of traversals overloading the system. With this system, it is not possible for few nodes to overload the network, because they should never have enough resources to do it.

The most interesting part of this algorithm is that it has a low overhead. In fact, it is possible to return only a subset of the routing tables. Furthermore, in average, there is no difference in term of node load between the centralized traversal and the sequential traversal presented previously (§ VII) because every node sends the same number of messages.

However, this methods has a cost. Every DST node is able to draw a map of the structure with the references of all the participants. This can be see as an offense of the user anonymity. Moreover, this can be used to pinpoint an attack against the structure.

IX. A Ring Hierarchy Algorithm

This algorithm, compared with the two previous sequential ones reduces the needed number of messages. Each group of a DST is a clique of its elements. Thus, a group can be used as a ring because its elements are also indexed. The Fig. 10 illustrates this approach similar to the hierarchical cliques interconnection network [8]. But in a DST there is no real intermediate nodes. So, it is possible to visit every node by only following the rings (Fig. 11.a).

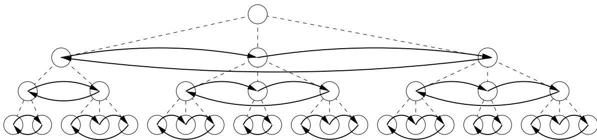


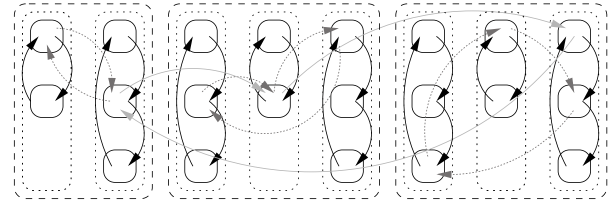
Fig. 10. A ring hierarchy

The fairness of the traversal is provided by the rings. The use of those rings is closely similar to the heuristic presented in the two previous traversals. The Fig. 11 illustrates a ring hierarchy traversal.

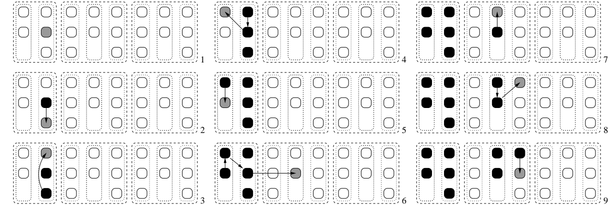
For every ring, this algorithm follows all its links. Thus the message is returned to the initiator or the node used as representative. The fact that the initiator of a ring walk received its message at the end of the walk is interesting for fault tolerance mechanisms. By receiving its message, the initiator is certain that every element of the ring has received the message and that the message was not corrupted in the process. This ring property is used by some protocols like Pilgrim [9] to assure the consistency of the system.

X. A Hamiltonian Path

A Hamiltonian path, also called a Hamilton path, is a path between two vertices of a graph that visits each vertex exactly once [10]. Build a DST Hamiltonian path do not require any processing. It is done by using the rings of the previous algorithm. The main difference with a DST Hamiltonian path



(a) d's view of the ring hierarchy



(b) the firsts steps of the traversal

Fig. 11. A ring hierarchy traversal initiated by d

is that the message is not returned to the representative element.

A Hamiltonian path traversal is done by sending the message $\langle c_1, c_2, ..., c_h \rangle$. This is a list of counters that stores the number of elements which as been visited in the ring of each level. The initiator sends the message $\langle 0, 0, ..., 0 \rangle$ to the next elements of its group of level 1. When a level 1 element received the message, it increments c_1. If c_1 is less than the number of elements in the group, it sends the message to the next element. If c_1 is equal to the number of elements of the current node group of level 1, it set c_1 at 0 and increments c_2. Then it send the new message to a node of the next element of its group of level 2. And so on.

This is possible because the groups are indexed cliques. Thus, every node knows how many elements are in its group for every level. And, every node knows a node member of the next elements of its group for every level. When c_1, c_2, ..., c_h are respectively equal to the number of elements of each level, the end of the path is reached. The Fig. 12 is an example of this traversal.

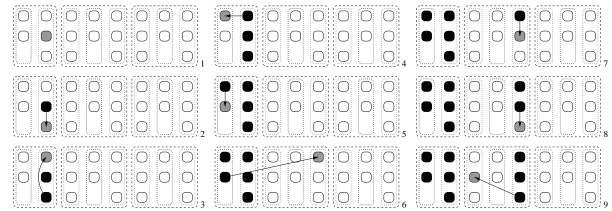


Fig. 12. A simple d's Hamiltonian path.

This Hamiltonian path is interesting in theory, but it has a significant drawback in practical use. A DST is not strictly balanced. Some groups have more nodes than others. Thus, in the whole DST, some nodes have more chance to be used as representative than others. This balancing breach is not significant in the other traversals because the lost of balance depends on the height of the DST which is in $\mathcal{O}(\log(n))$.

But in the Hamiltonian path, there is no comeback that permit to erase the lost of balance. Then the lost of balance is accumulated from step to step. Practical uses show that there is a convergence of the paths and a subsets of nodes got more visits than the others when partial traversals are done.

A solution is to add some salt to destroy the convergence. Instead of visiting one element after the others by following their index number, the elements of a group are visited randomly. The state of a element – it is already visited or not – can be stored with a flag. So, the space needed to store the element list which are already visited or not, takes few memory space and can be easily sent with the message.

Thus, when a node receives the message, it picks randomly an element of its level 1 that was not already contacted. If all the element of level 1 are already visited, it picks randomly an element of level 2 that was not already contacted. The resulting representative will start a level 1 walk by picking randomly an element of its level 1. And so on. This second Hamiltonian path destroys the convergence observed in the first one. The fairness rule is honored. The Fig. 13 shows one of these Hamiltonian paths.

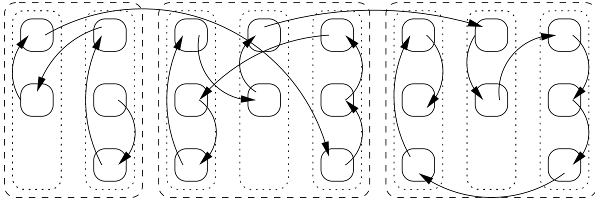


Fig. 13. A fair d's Hamiltonian path.

XI. The Hidden Faces of the DST

The last algorithm outlines an hidden faces of the distributed spanning tree. This last section discuses about them and explains which aspects are easy to implements and where are the difficult parts.

A. Balance of the DST

The DST is balanced like a red-black tree. A strictly balanced tree can be done if the total number of nodes is known in advance. This is also possible if the DST structure is rebuild each time that the number of node change.

But those optimal cases are not realist. Most of the time the number of nodes is unknown and best effort algorithms are needed to keep the load balanced. The DST structure itself assures that no groups of level i has more than b^i nodes and less than a^i nodes.

Some heuristics are also available to populate in priority small groups. Picking a group randomly is not a good one because groups with more groups than the others will get more nodes a become bigger. A better approach is to pick a node – randomly or not –, and to go down the tree by choosing randomly an elements each time we go one step above. The resulting leaf is chosen and the new node is inserted into its group of level 1. Like that, each group has the same probability to get new nodes.

Another method is to transfer some nodes from a group to another if the DST becomes unbalanced. But this method has the two following drawbacks: it need some global knowledge to know which groups are more loaded than the others; and a ping pong effect can appear when groups of nodes are exchanged periodically by two groups.

B. Faults Tolerance

It is assumed that the DST has strong faults tolerance properties. But it was not proved formally. The cliques are very tolerant toward node failures. But an assembly of cliques can be frail if the glue which maintains the cliques together is weak. This is not the case of the DST because the glue is inside every node. Each node links together its groups of every level and this glue is only broken when the node fail.

In our implementation, the fault detection is done in the groups of level 1. A and B are any two nodes of a same group of level 1. If A does not received any message from B after a certain period of time, A initiates the algorithm that removes B from the DST. This algorithm does not need that B is alive to be run. When the activity is low, ping messages are sent to avoid accidental removing of nodes.

This document does not explain what happens if a fault is discovered when a traversal is run. This is intentional because the real question is: what does a traversal means when nodes come and back in the structure randomly. The answer is closely dependent of the implemented system and not of the DST structure.

Checking the consistency of a DST is easy. Each time a message is sent to an element, the sender puts inside the message its knowledge about the group. This is the number of elements in the group, its virtual name and the group virtual name. Only few bytes are needed. The addressee checks those information to see if they are consistent or not. Dealing with inconsistency is difficult. If an inconsistency is detected, the other elements of the group are contacted to know who is right. If no solution is found, the whole structure is disconnected and rebuild from scratch.

C. Implementation Difficulties

The presented algorithms are all relatively easy to implement. Adding and removing a node is also easy. The insertion and deletion algorithm are based on the B-trees algorithms [11].

When a group is full and there is a need to add a new element, this group should split into two groups. When a group as the minimum required number of elements and an element leaves the group, the group should merge with another group. This is easy to implement in centralized context but not in a distributed context.

Mutexes are used in this algorithm. Each group has its own mutex. The mutex is the first element of the group. Every element of a group knows the elements order, so it can contact the mutex easily. Each mutex are doubled for fault tolerance issues. The split or the merge are done by multicasting the order to all the concerned nodes.

What's append if only a part of the concerned nodes received the message and not the other due to a fault ? The DST become inconsistent and can not be repaired. This a unresolved issue.

XII. CONCLUSION

The main originality of the distributed spanning tree is that the intermediate nodes of the trees are replaced by groups of nodes. Classify elements in groups and groups of groups is not new. Zoologists did it from the antiquity. The object oriented programming and the Internet Protocol address space also use this technique. TOPLUS [12] also specify a classification in groups of groups to optimize the network use for the distributed hash tables. But using those groups as tree intermediate nodes without creating bottlenecks was the challenge of the DST.

Plenty of structures were studied to build parallel computers. There are all interesting because they efficiently organize the nodes. They address bottlenecks and fault tolerance. Unfortunately, few structures are available for large scale system and the random graph seem to be the better solution. A notable exception is the distributed hash tables. But the distributed hash tables can not do everything. The DST adds a new element to the scalable overlay network list.

The section III explains that the DST has some properties describes as the equiprobability rule and the hierarchical structure. The hierarchical structure is used to do efficient traversals, and the equiprobability rule helps to avoid bottlenecks. All the described algorithms use at most $2i$ messages to query i nodes. It is also possible to send a message to i nodes in parallel or by following a Hamiltonian path with only i messages.

But those properties do not provides fairness alone. The traversal algorithms can not do whatever and enjoy the fairness property of the DST. The fairness property means that if only a partial traversal is done, every node has the same probability to be visited. This is of primary importance for discovery mechanisms. If the traversal is not equitable some nodes are found more often than the others, so these nodes receives more load than the others.

Seven DST traversal algorithms have been presented. The parallel traversal is used to send efficiently a message to every node. The breadth first search is used to query in parallel a number of node which grows exponentially with the number of steps. Then the limited depth parallel traversal limits the number of nodes contacted in parallel and pursues the traversal with a sequence of small flooding. After, the first sequential traversal was presented. Then the centralized algorithm shows that it is possible to put all the works of a traversal on its initiator. Finally, the ring hierarchy was used to introduce some Hamiltonian paths.

The distributed spanning tree is a scalable structure. But this is not the case of its algorithms and great care should be take when using them in large scale systems. If nodes use an unbounded traversal to locate rare items, then lot of messages are sent and the system collapse under the load. But if multicast mechanisms are implemented, the DST can be used. However, If the multicast is one source only, the DST behaves as a simple tree and specialized multicast tree should be use instead.

The DST has been tested in small systems and simulations have been made for large scale systems. However, no real use of the DST in large scale system has been done. So, it is not known if another hidden factor will show up and if the system will degenerate in a not obvious way in large scale.

REFERENCES

[1] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A survey and comparison of perr-to-peer overlay network schemes," *IEEE Commun. Survey*, submitted for publication.

[2] X. Li and J. Wu, "Searching techniques in peer-to-peer networks," *Handbook of Mobile Computing*, accepted to appear. [Online]. Available: http://www.cse.fau.edu/~jie/research/publications/Publication_files/p2psearching.pdf

[3] S. Dahan, J.-M. Nicod, and L. Philippe, "The distributed spanning tree: a scalable interconnection topology for efficient and equitable traversal," internal report, Dec. 2004. [Online]. Available: http://lifc.univ-fcomte.fr/publis/pub/2004/RR2004-17.pdf

[4] T. Klingberg and R. Manfredi, "Guntella 0.6," RFC draft, June 2002. [Online]. Available: http://groups.yahoo.com/group/the_gdf/files/Development/

[5] (2000) Gnutella: To the bandwidth barrier and beyond. Clip2 DSS. [Online]. Available: http://lambda.cs.yale.edu/cs425/doc/gnutella.html

[6] M. Zaitsev and P. Verdy. (2004) Mutella is abusing the GWebCache network. forum. [Online]. Available: http://sourceforge.net/forum/forum.php?thread_id=823417&forum_id=114921p

[7] M. Satyanarayanan, "The influence of scale on distributed file system design," *IEEE Trans. on Software Eng.*, vol. 18, no. 1, Jan. 1992.

[8] S. Campbell, M. Kumar, and S. Olariu, "The hierarchical cliques interconnection network," *Elsevier J. Parallel Distrib. Comput.*, vol. 64, pp. 16–28, Jan. 2004.

[9] H. Guyennet, J.-C. Lapayre, and M. Tréhel, "Distributed shared memory layer for cooperative work," IEEE Computer Society and TC Computer Communications, Minneapolis, USA, pp. 72–78, Nov. 1997.

[10] E. W. Weisstein. Hamiltonian path. web page. MathWorld. [Online]. Available: http://mathworld.wolfram.com/HamiltonianPath.html

[11] D. E. Knuth, *The Art of Computer Programming*. 75 Arlington Street, Suite 300, Boston, MA 02116: Addison-Wesley, 1998, vol. 3, ch. 6.2.4.

[12] L. Garcé-Erice, E. Biersack, P. Felber, K. W. Ross, and G. Urvoy-Keller, "Hierarchical Peer-to-Peer Systems," in *Proceedings of the 9th International Euro-Par Conference on Parallel Processing (Euro-Par 2003)*, 2003, pp. 1230–1239.