The Distributed Spanning Tree Structure

Sylvain Dahan, Laurent Philippe, Jean-Marc Nicod

▶ To cite this version:

Sylvain Dahan, Laurent Philippe, Jean-Marc Nicod. The Distributed Spanning Tree Structure. IEEE Transactions on Parallel and Distributed Systems, 2009, pp.1738–1751. hal-00560821

HAL Id: hal-00560821
https://hal.science/hal-00560821

Submitted on 30 Jan 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

The Distributed Spanning Tree Structure

Sylvain Dahan, Laurent Philippe, and Jean-Marc Nicod

Abstract— **Search algorithms are a key issue to share resources in large distributed systems as peer networks. Several distributed interconnection structures and algorithms have already been studied in this context. With expanding ring algorithms, the efficiency of searches depends on the topology used to send query requests and on the dynamics of the structure. In this paper, we present an interconnection structure that limits the number of messages needed for search queries. This structure, called Distributed Spanning Tree (DST), defines each node as the root of a spanning tree. So, it behaves as a tree for the number of messages but it balances the load generated by the requests among computers and, thus, it avoids to overload the root node. This structure is scalable because it only needs a logarithmic memory space per computer to be maintained. A formal and practical description of the structure is presented and we describe traversal algorithms. Simulations show that DST based searches behave better than randomly generated graphs and trees as it generates less messages to query all computers while avoiding the tree bottlenecks.**

Index Terms— **Distributed systems, Search algorithms, Expanding Ring Algorithms, Interconnection graphs.**

I. Introduction

TWENTY years ago, networks have seen tremendous advances in computer topologies. Hypercubes, tori and rings are classical examples. At this time, the goal was, on one hand, to build a topology that achieves the best bandwidth between processors and tolerates faults. On the other hand, it tries to minimize the number of links because computer buses, that implement those links, were expensive. Nowadays, overlay networks benefit from the interest on topologies even if the context is different: links are no longer expensive because the cost of opening a TCP/IP link is negligible and the number of peers is dynamic — we no longer have n^2 processors. Trees, connected graphs and Distributed Hash Tables (DHT) [1] are common overlay network topologies.

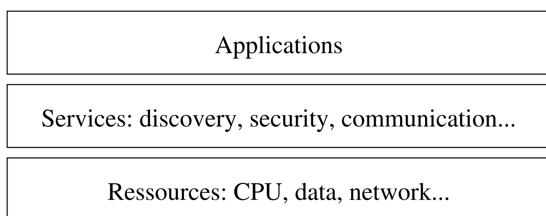


Fig. 1. Multi-layered Architecture.

As for physical networks, an overlay network topology is the lower layer of a multi-layered system (see figure Fig. 1). Middle layers — usually offered by middlewares — are composed of services, as discovery, group management or security. Upper layers gather applications, as resource sharing or virtual communities. The implementation and efficiency of these services and applications closely depend on the characteristics of the overlay network. However, the link between the overlay network and the services is not so close as the link between communication protocols and physical networks. Different services may used different overlay networks. The Distributed Spanning Tree presented in this paper is relevant to discovery and broadcasting algorithms. Other topologies as the DHT are more relevant to identify and access data stored on the overlay.

Contact:laurent.philippe@lifc.univ-fcomte.fr. Laboratoire d'Informatique de l'Université de Franche-Comté, F-25000 Besançon, France.

Search algorithms for distributed systems is an old and well known computer research field. Even if peer-to-peer search algorithms like Gnutella and Distributed Hash Tables (DHT) have bring some novelty, the basic issue stays the same: look-up in a directory versus direct resources querying. There is a great difference between these two approaches. (i) Looking up in a directory implies to link a structure to the searched information. Universal Description, Discovery and Integration (UDDI), Jini, Lightweight Directory Access Protocol (LDAP), Domain Name System (DNS) and DHT are implementations that relay on an information structure. Looking up in a directory is appropriate to distribute and efficiently access informations on a peer-to-peer network. This implies a centralized view of the system. (ii) Direct resource querying relays on no structure except the connections between peers. Gnutella, Address Resolution Protocol (ARP) and NetBIOS name service are examples of this approach Direct resource querying fits to locate resources in self organized peer-to-peer networks, without a centralized view. In this case, the resources are not placed according to an index but rather published by their owners on the network and made available from their computer.

In this paper we focus on direct resource querying. We aim at efficiently discover, search or broadcast on a peer-to-peer network where nodes can dynamically come or leave at any time. In this context, a popular direct querying algorithm is called expanding ring, or TTL-based flooding. With this algorithm, resources are connected through a graph and the algorithm forwards queries through the graph to find a particular resource. With expanding ring algorithm, it is well known that search performances are affected by the communication graph topology. The structure of the Distributed Spanning Tree, described in the following, provides better performances than usual topologies as trees or random graphs.

To show our contribution on this context we organize the paper as follow. In the section II, to take a stand on the network structure domain, we present related works on topologies and overlay networks. The context and the originality of our contribution is described in the third section. The section IV defines the structure and the properties of the DST. Traversal and construction algorithms are described in the section V. Then, to validate our work, we present in the next section the results of simulations that compare DST, tree and graph topologies. Lastly, we discuss and conclude with few DST's issues.

II. Related Works

The Distributed Spanning Tree is not the first structure that tries to interconnect a set of nodes efficiently. We can distinguish three main classes of interconnection structures. (i) Static structures like hypercubes or fat-trees were studied extensively and are used to build parallel computers. (ii) Dynamic structures like Distributed Hash Tables are organized, distributed and specifically designed to store and access data or resources in distributed systems. (iii) Finally, self-organized structures do not relay on a strict organization but provide a support to efficiently communicate between peers or interconnect virtual communities. Instances of these three classes of structures are presented in the remaining of this section. Most of them are not suitable for our context however they may be used as a source of inspiration to interconnect peers.

In [2] S. Campbell, M. Kumar and S. Olariu proposed a static structure called hierarchical cliques (HiC). The HiC is a k-ary tree, modified to enhance local connectivity in a hierarchical, modular fashion. The k children of every node are grouped together to form a clique. These cliques add robustness and alternate paths to the tree structure. This topology was designed to build parallel computer which combines the positive features of the tree and the hypercube. But in the HiC, all nodes are not equal. There are processor elements that are the leaves and switching elements that forward the messages. Due to the static structure of the HiC, if the root and its k children fail or leave, the whole structure is split in k separated groups.

Q. M. Malluhi and M. A. Bayoumi [3] proposed the Hierarchical Hypercube (HHC). An n-HHC has three levels with $n = 2^m + m$. The first level is constituted of 2^n nodes. At the second level the 2^n nodes are grouped into 2^m hypercubes of 2^m nodes called SonCubes. At the third level, the 2^m SonCubes are connected in an hypercube fashion to form a father cube. Each SonCube has exactly 2^m links that connect it to the other SonCubes, and each link is incident to one node of the SonCubes. This is interesting because every node has the same role. It also implements an interesting vision: it uses recursion on a distributed structure to add several levels and it is able to keep a degree for each node with a complexity order of $O(\log(n))$. But the HHC is designed to build parallel computers and not to build distributed systems. So, it has a low degree for each node and needs a static number of nodes. It is also a very static structure where it is difficult to remove or to add a node.

In [4] S. D. Gribble *et al.* introduced a dynamic data structure (DDS) with a Distributed Hash Table (DHT). The DHT provides incremental scalability of throughput and data capacity when more nodes are added to the cluster. To achieve this, they horizontally partition the table to spread operations and data across bricks. The DDS was used to build a large scale file service. Then the DHT were integrated in peer-to-peer systems where they gain more scalability. Chord and Pastry are two DHT implementations. A. Rowstron and P. Druschel [5] explain that Pastry bears some similarity to the work by Plaxton *et al.* [6]. In the Plaxon structure, each object has an unique address x of n bits. The structure uses the address prefix to route the message to the object. Every node has a routing table of $\frac{n}{b}$ levels. The ith level of the routing table is a list of links that satisfy the following constraints: (i) the $(i-1).b$ bits prefix of a pointed node must be the same as the current node, (ii) for the 2^b permutations of the $i.b$ bits prefixes with the same $(i-1).b$ bits prefix two nodes *(for fault tolerance)* are pointed by the routing table. Using this structure, at most $\frac{n}{b}$ hops are needed to route a message to an object. The approach of routing based on address prefixes can be viewed as a generalization of the hypercube routing. As an interconnection structure, the DHT are interesting because every node is able to send a message to an other node in only $O(\log(n))$ hops and they just needs $O(\log(n))$ entries in its routing table. They are also theoretically scalable and resistant to failures. However the index implementation of a DHT lays on a global view of the system and determines the data placement whereas in self organized networks, every peer is independent and provides its own data or resources from its computer to the network.

Some peer-to-peer systems use less structured topologies and use best effort algorithms. The Gnutella specifications [7] explains that a Gnutella servant connects itself to the network by establishing a connection with another servant currently on the network. The acquisition of another servant's address is not part of the protocol definition. So the topology is a connected graph without a more precise specification. In such topology, search and multicast are done by flooding. Despite the high bandwidth consumption of the flooding, the structure is strong against faults and the algorithms that add or remove a node are straightforward. The Kazaa networks allow to pass an additional scale by the use of supernodes [8]. Nodes elect supernodes to represent themselves. The discovery is done between supernodes and regular nodes are not longer queried by the discovery process. It results in a reduction of message emissions. JXTA [9] uses a similar method, although supernodes are called rendez-vous peers. JXTA also adds a cache mechanism to achieve better performances. Finally, this supernodes method is implemented in the discovery mechanism of the Distributed Integrated Engineering Toolbox (DIET) [10], a RPC-GRID middleware based on the Application Service Provider paradigm. Supernodes decrease the global communication load but they are not scalable, as they just decrease this load of one order of magnitude, nor suited for self-organized networks, as they must be administrated.

Few systems try to share the communication load between the peers like SplitStream [11] and BitTorrent [12]. B. Cohen introduces BitTorrent as a file distribution system with Pareto efficiency. A BitTorrent platform is composed of a tracker and peers. The tracker is a directory with the reference of all the peers which replicate a file. With the help of the tracker, each peer connects itself with twenty or forty other peers. Peers exchange pieces of the data with a "tit-for-tat" policy. The system is able to achieve good performances and scalability by using a simple best effort algorithm and by sharing the load between the peers efficiently. M. Izal *et al.* [13] describe the performance of BitTorrent over a five months period. Unfortunately, the latency is high and BitTorrent is dedicated to data sharing. So it cannot be used to share resources and access services.

III. Context and Contribution

In this section we present first the context which justify our work and then our contribution.

A. Project Context

The Distributed Interactive Engineering Toolbox (DIET) [14], [15] is a grid middleware which allows clients to access application servers via Remote Procedure Calls (RPC) [16]. For each remote call, DIET tries to find the best suited server by taking into

account, the application availability, the server load, the location of processed data and the average time used by the server to process similar calls. Because most of those informations are highly dynamic, it is more efficient to query every server at each request than to store or cache all those informations in a directory and update them regularly.

To allow efficient communications, the DIET architecture is based on a broadcast tree (figure Fig. 2). The root of the tree is called Master Agent (MA) which is the front end between a DIET platform and clients. Application Servers (SeD) compose the leaves and Local Agents (LA) forward queries from the Master Agent (MA) to the Application Servers (SeD). Because it is useless to query a server that does not support the procedure searched by the client query, a filter has been implemented: each Application Server provides its parent node (LA or MA) with the list of its applications. Then, the parent forwards to its own parent the list of the software available in its descendants, and so on. Because every query lists the software that it searches, it is easy to not send messages in branches that do not have the needed software. Experimental studies show that this tree architecture is efficient and allows a good scalability in term of requests frequency [17].

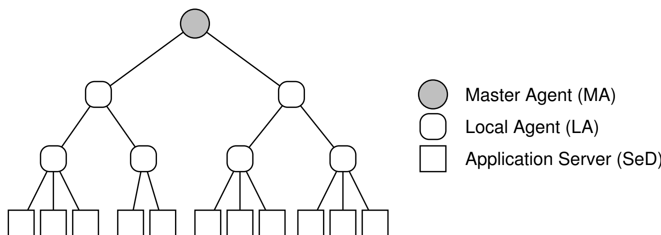


Fig. 2. The DIET architecture.

However, this architecture has a scalability limitation in term of number of servers and clients as every request is sent and managed by the MA node. Moreover, for each query, every server that owns the needed software is queried. When there are lots of queries, servers are overloaded by queries. This is pointless because a client does not want to access the best server of the world. He only wants to access a good server. So, it is preferable to just contact a subset of the available servers than the whole set of servers. For this reason, the whole set is split in several small trees which each recover a limited area. All these subtrees are considered as independent peers and the whole set of peers constitute a self-organized community. We added this mechanism into DIET by implementing a Gnutella like flooding algorithm where there are several small DIET platforms which are connected by a communication graph through their Master Agent [10]. If a Master Agent does not find an available server, then it forwards the query to its neighbors, and so on.

We can explore several ways to optimize the expanding ring algorithm. We can use caching and routing mechanisms. We can optimize the graph by tuning its specifications: which degree? Should it be scale-free? Should it be small-world? Semantical search algorithms are also promising ways to optimize it. Finally, we can try to find a better suited topology that the usual graph. This topology is presented in the remaining of this paper. It was design to support large DIET platforms but it is also suitable for interconnecting self-organized overlay networks.

B. Contribution

We propose a new topology, called Distributed Spanning Tree (DST), that optimizes flooding algorithms — TTL-based search algorithms — for search applications to get better search performances. The idea comes from the opposition between tree and graph topologies[1]. Tree topologies are interesting because of their complexity in number of messages for querying (request-reply) nodes: only $2n$ messages are needed for n computers. However, if each tree node is a computer, then the tree topology suffers from bottlenecks as the root node will concentrate most of the messages. Graph sends more messages but does not suffer from any bottleneck, as the load of messages will be shared between the nodes. Thus, in practice, graphs are more efficient than trees.

The tree-based overlay networks suffer from a major drawback when the computer behavior is determined by its identity. Indeed, the root computer initiates a search and sends a message to its children. Intermediate node computers — computers that are not a leaf and not the root — wait for messages and forward them to their children and finally, leaf computers wait for messages. The unbalance is obvious, most computers are usually leaves (if the tree is balanced) and they do nothing in regard to message forwarding.

Conversely, it is possible to create a tree where every computer behaves at the same time as a leaf, as the root and as intermediate nodes and where the root node is distributed between the whole set of computers. This is exactly what the Distributed Spanning Tree (DST) does: each computer is a leaf and each non-leaf node is distributed through its children. Each computer is the root of its own spanning tree.

IV. Structure definition

A DST can be described at three different levels. The logical level is an abstract vision of the DST. At this level, tree nodes — that are groups of computers — are linked together by abstract links. Then comes the interconnection level that implements inter-nodes links with TCP/IP links. Finally, there is the topological level which describes how the TCP/IP links map on a real network.

A. Logical Level

1) General Idea: If we have 5 computers and want to build a tree of arity greater than 4, we have two solutions. The first is obvious. Take one computer randomly and link it with the other. This computer becomes the tree root and at the same time, it becomes the bottleneck of the tree.

The second solution challenges the common assumption that a node must be a computer. 5 computers is not a lot. We can consider in practice that the 5 computers know each others. In other words, we have a complete graph and this complete graph is our root node. This is important because saying that a parent node is the complete graph of its children is the DST's fundamental concept.

By being a complete graph, the root is inherently distributed. When the root wants to send a message to its children, it elects

[1]In this document, we use the term of graph as a shortcut for random undirected graphs or more precisely pseudo-random undirected graphs because these communication graphs are not truly random: their algorithms usually bound the number of links between a node and the others and they usually assume or guarantee that the graph is connected.

one of its children and orders it to send the message to its brothers. Doing it is possible, thanks to the complete graph.

2) Description: As a tree, a DST is a recursive structure. A DST has usually several stages where each stage is built with elements of the above stage:

- **Stage** 0 contains the leaves of the DST. Every leaf is a computer and every computer that participates to a DST must be a leaf.
- **Stage** $n+1$ (with $n > 0$) contains the parents of the stage n nodes. Each parent is a complete graph of its children and the number of children is bounded by a and b where $0 < a \leq \frac{1}{2}b$.
- **Stage** h (with h the DST height), is different from the others in the fact that it contains the DST root and that the root can have less than a children.

The following figures illustrate the DST structure at the logical level.

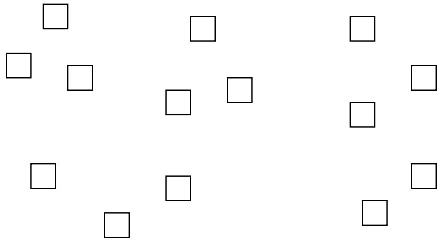


Fig. 3. Logical structure, stage 0.

Figure Fig. 3 displays the computers set that constitutes a DST. Every computer is drawn with a square. Every square is also a DST leaf node.

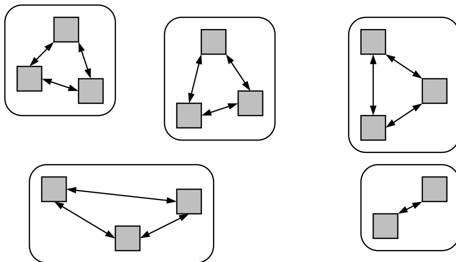


Fig. 4. Logical structure, stage 1.

Figure Fig. 4 displays the stage 1 nodes. Those nodes — white round squares — are composed of small complete graphs of leaves — gray squares. The arrows represent the logical links between the leaves that form the complete graphs.

Figure Fig. 5 displays the stage 2 nodes. Those nodes — white round squares — are composed by small complete graphs of stage 1 nodes — gray round squares. The arrows represent the logical links that form the complete graphs. Those links are abstract and do not directly connect computers but they logically connect stage 1 nodes together. How those links are implemented is defined by the interconnection level.

Figure Fig. 6 displays the stage 3 node, root of this DST. The root node — white round square — is composed by a small complete graph of two stage 2 nodes — gray round squares. The arrows represent the logical links which form the complete graph.

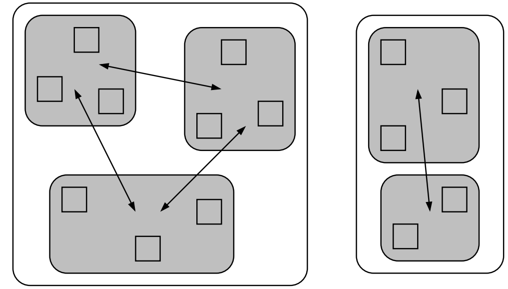


Fig. 5. Logical structure, stage 2.

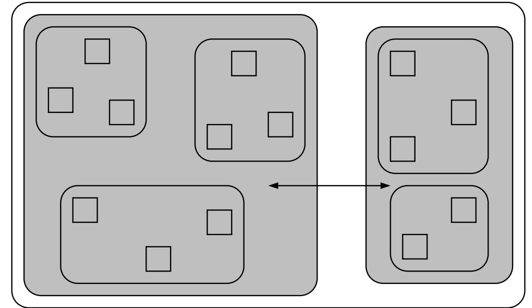


Fig. 6. Logical structure, stage 3.

3) Invariants Definition: To be a DST, a graph must respect several invariants. This paragraph describes the seven invariants that define the DST logical level. The following definitions are used in the expression of invariants:

- h is the DST height,
- $\mathcal{C}$ is the set of computers that participate to the DST,
- $\mathcal{S}_n$ is the set of nodes that form the stage n of a DST, with $0 \leq n \leq h$,
- X, Y, Z are nodes of the DST. A node X is a graph $X(V, E)$ where V is the set of vertices, the children nodes, and E is the set of logical links that interconnect these nodes.
- $d(X, Y)$ is a link between two nodes X and Y,
- u, v are computers that participate to the DST,
- the root node is noted ϵ.

Firstly, the children of a stage 1 node are computers.

Invariant 1: For each node X of $\mathcal{S}_1$, every element of V is also an element of $\mathcal{C}$, the computers set.

$$X(V, E) \in \mathcal{S}_1 \wedge u \in V \Rightarrow u \in \mathcal{C}$$

Secondly, every computer that participates to the DST must be a child of one and only one stage 1 node.

Invariant 2: For all node $X(V, E)$ of $\mathcal{S}_1$, the union of the V sets is equal to the computer set and every intersection of two nodes is the empty set.

$$\bigcup_{X(V,E) \in \mathcal{S}_1} V = \mathcal{C} \wedge$$

$$\forall X_i(V_i, E_i), X_j(V_j.E_j) \in S_1 \Rightarrow X_i(V_i, E_i) \cap X_j(V_j.E_j) = \emptyset$$

So, from this invariant we can note that $\mathcal{S}_0 = \mathcal{C}$.

Thirdly, the children of a stage n node, which is not a computer, are stage $n-1$ nodes.

Invariant 3: For a node X of $\mathcal{S}_n$, where n is not equal to zero, a child u of the V set belongs to $\mathcal{S}_{n-1}$.

$$n > 0 \wedge X(V,E) \in \mathcal{S}_n \wedge Y \in V \Rightarrow Y \in \mathcal{S}_{n-1}$$

Fourthly, every stage n node that is not the root node must be the child of a stage $n+1$ node.

Invariant 4: For all stage n node Y where stage n is not the root stage, there is a stage $n+1$ node that is the parent of Y.

$$n < h \wedge Y \in \mathcal{S}_n \Rightarrow \exists X(V,E) \in \mathcal{S}_{n+1} \mid Y \in V$$

Fifthly, a node is the child of a unique node.

Invariant 5: For all two nodes $X(V,E)$ and $Y(V',E')$ of the same stage n, where n is not zero, there is no node that can be a child of these two nodes. So no stage $n-1$ node belongs at the same time to V and V'.

$$n > 0 \wedge X(V,E) \in \mathcal{S}_n \wedge Y(V',E') \in \mathcal{S}_n \wedge X \neq Y \Rightarrow V \cap V' = \emptyset$$

Sixthly, the number of children for a node is bounded by two values a and b.

Invariant 6: A stage n node X must have a number of children that is equal or less than b. The number of its children must also be greater that a if X is not the last stage node, the root node.

$$X(V,E) \in \mathcal{S}_n \Rightarrow |V| \leq b \wedge (n = h \vee |V| \geq a)$$

Finally, stage n nodes are complete graphs made up of stage $n-1$ nodes.

Invariant 7: A stage n node X, where n is not zero, is a complete graph of its children. So for two nodes u and v of V, there is a link in E that connects them.

$$n > 0 \wedge X(V,E) \in \mathcal{S}_n \wedge Y \in V \wedge Z \in V \Rightarrow d(Y,Z) \in E$$

B. The interconnection level

1) Description: The logical level is an abstract vision of the DST. The interconnection level is the implementation of this vision. It describes how nodes are distributed through computers and how computers implement inter-node abstract links on TCP/IP connections.

Non leaf nodes are complete graphs of their children, so they are inherently distributed among their children. Recursively, non leaf nodes are distributed among their descendants. At the end, every non leaf node is distributed among computers because all leaf nodes are computers.

Logical level abstract links of level 1 are implemented by linking all computers of a node together to form a complete graph. Then, for upper level, if there is an abstract link between two nodes A and B, then every computer that is a descendant of A opens a TCP/IP link with one computer that is a descendant of B and every computer that is a descendant of B opens a TCP/IP link with one computer that is a descendant of A. In this way, every computer of A can directly send a message to B and vice versa.

Even if the DST structure does not care about how computer pairs are constituted, this choice has an impact on the DST performances. The details depend on the application and their priorities. However, if there is an abstract link between A and B, we do not recommend that every computer of node A opens a link with the same computer of B. To allow better fault tolerance and load distribution, the algorithm should take care that computers of A are linked with different computers of B as shown in the figures Fig. 8 and Fig. 9.

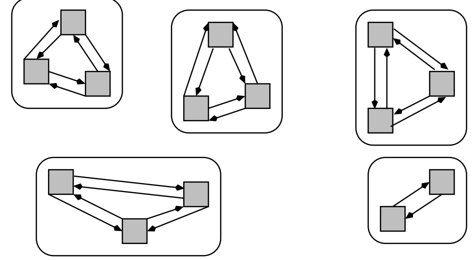


Fig. 7. Interconnection level, stage 1.

Implementing stage 1 abstract link is simple because stage 1 nodes are complete graphs of computers. Figure Fig. 7 shows the TCP/IP links that form the stage 1 nodes.

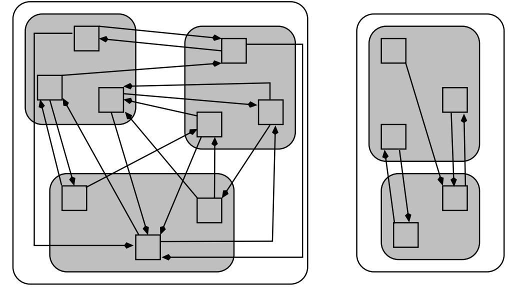


Fig. 8. Interconnection level, stage 2.

Figures Fig. 8 and Fig. 9 show the TCP/IP links that form respectively stage 2 and stage 3 nodes. You can see that for each logical level abstract link defined on figures Fig. 5 and Fig. 6, every computer of a node is connected to one computer of the other node and vice versa.

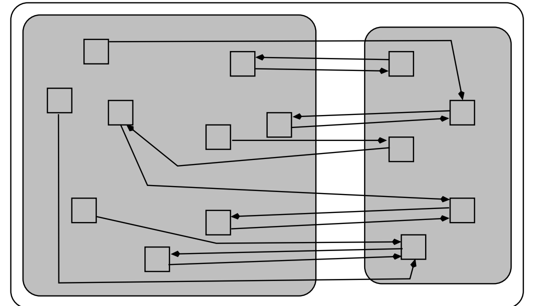


Fig. 9. Interconnection level, stage 3.

For each level, a computer is also connected to the node it belongs to by being connected to itself.

2) Invariants Definition: The interconnection level adds few invariants to the DST specifications. At the interconnection level, we use the same definitions than for the logical level and we add $\mathcal{L}$ the TCP/IP links set that forms a DST.

Firstly, we need to define what is a descendant.

Invariant 8: X is a descendant of Y means that X is a child of Y or the node X is a descendant of a child of Y.

$$desc(X, Y(V, E)) \Rightarrow X \in V \vee (Z \in V \wedge desc(X, Z))$$

For each abstract link that connects two nodes, every computer of a node opens one TCP/IP connection with one computer of the other node and vice versa.

Invariant 9: For each abstract link that connects the two children Y and Z of a node X there is a TCP/IP link $d(u, v)$ that connects one descendant u of Y with one descendant v of Z.

$$n > 0 \wedge X(V, E) \in \mathcal{S}_n \wedge Y, Z \in V \wedge d(Y, Z) \in E \Rightarrow$$

$$(\exists u \in \mathcal{C} \mid desc(u, Y)) \wedge (\exists v \in \mathcal{C} \mid desc(v, Z)) \wedge d(u, v) \in \mathcal{L}$$

And we specify that there is no link that does not correspond to invariant 9.

Invariant 10: For each TCP/IP link $d(u, v)$ of a DST, there is an abstract link $d(Y, Z)$ that connects two children Y and Z of a node X of a stage n where u is a descendant of Y and v is a descendant of Z.

$$u \in \mathcal{C} \wedge v \in \mathcal{C} \wedge d(u, v) \in \mathcal{L} \wedge n > 0 \wedge X(V, E) \in \mathcal{S}_n \Rightarrow$$

$$(\exists Y, Z \in V \mid desc(u, Y) \wedge desc(v, Z))$$

Finally, every computer is connected to itself.

Invariant 11: Every computer u of $\mathcal{C}$ is connected to itself.

$$u \in \mathcal{C} \Rightarrow d(u, u)$$

3) Implementation: All these TCP/IP links are managed by every computer thanks to a routing table. To justify these routing tables, we first need to define two naming notations:

1) Each computer connected to a DST has an IP address. For clarity, we represent IP addresses with a single character written in a different font. For example, *a*, *b* and *c* represent the IP addresses of three computers.
2) Every node is indexed. Figure Fig. 10 shows how the nodes are indexed. The name of the root node is the empty string ϵ. The children of a node are indexed with a number that ranges from 1 to the number of children. Then, the name of a children is formed by the concatenation of its father name and its index. For example, the root node of figure Fig. 10 has two children: 1 and 2. Then the node 1 has three children: 11, 12 and 13.

At the interconnection level, each computer knows one computer for each of its brothers at every stage. This knowledge is stored in a routing table which matches the computer's local view of the DST. Table I displays the routing table of computer *e*.

Each line of this table stores data for a different stage of the DST. The first column indicates which stage is represented by this data. The second column indicates, for this stage, the index of the child that contains the current computer *e*. It can be noted that we obtain the name of the leaf that contains the current computer *e* by reading the second column from top to bottom. The following columns indicate for each child, which computer is used as the child's representative level. Each time the computer *e* wants to contact a child, it sends its message to the computer whose address is indicated in the routing table.

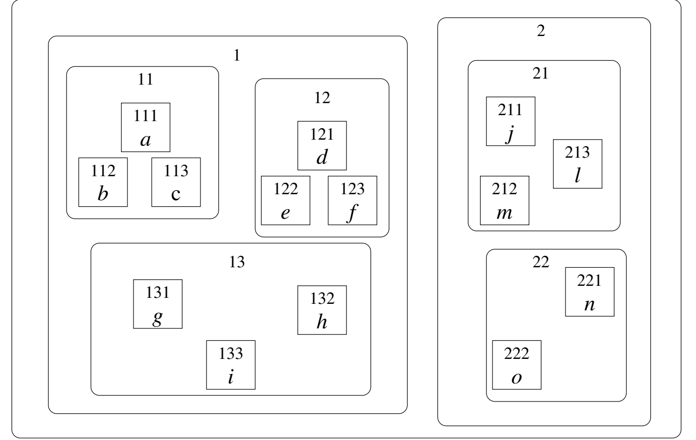


Fig. 10. DST structure with node names and computer's IP addresses..

TABLE I
COMPUTER *e*'S ROUTING TABLE.

position		representatives		
stage	index	1	2	3
3	1	*e*	*m*	–
2	2	*c*	*e*	*i*
1	2	*d*	*e*	*f*

The routing table is the only data structure that is needed to store the DST structure. We can conclude that the memory complexity order for an n-computers DST is $\mathcal{O}(\log(n))$ per computer because the routing table size is $b \times h$, where h is the number of stages and b the upper bound of the number of children of a node: hence the following formula $\log_b(|\mathcal{C}|) \leq h \leq \log_a(|\mathcal{C}|) + 1$ (see theorem 2 in the section IV-D).

C. The topological level

The topological level defines how the TCP/IP links should be mapped onto a real network. Studies about this level are still at the beginning but give an idea of what is possible to do. We describe two examples to show how a DST can be mapped onto the Internet and on a semantical network.

1) Mapping onto the Internet: The Internet, as an interconnection of networks, has a hierarchical structure. Local Area Networks (LAN) are the lower stones of the Internet. Those LANs are aggregated to form sites like university campuses or Metropolitan Area Networks. Those sites are then interconnected in an autonomous system and autonomous systems are interconnected together to form the Internet.

Message transfers are more efficient and inexpensive if they only pass through a LAN, and become slower and more expensive when they go through a site, an autonomous system or through the Internet. Thus, it is desirable to encourage local communications and to limit long distance communications.

In average, we observed that there is $\frac{a+b}{2}$ more messages exchanged between the *stage* n nodes than between *stage* $n+1$

nodes because each *stage* $n+1$ nodes has $\frac{a+b}{2}$ children, in average. Thus, it is interesting to map a DST hierarchy onto the Internet one. Computers that belongs to the same LAN are grouped together to form low level nodes. Then, low level nodes that belong to the same site are grouped together and so on. This way, the structure favors the use of local communications.

2) Mapping on a semantical network: Some TTL-based search mechanisms classify resources semantically [18]. They group together resources that share common interests. This is based on the hypothesis that the group has a higher probability to provide the searched resource when the resources/users that compose it share a limited number of interests. This leads to more efficient searches as less resources/users are queried during a lookup request.

One of those mechanisms, the Dynamic Publish/Subscribe (DPS) system [19], [20], organizes resources with a semantical tree but suffers from bottlenecks. Using a DST, to map the DST groups onto the semantical groups, keeps the hierarchical classification while removing bottlenecks and thus allows to keep the optimization proposed by the DPS.

D. Height of a DST

In this section, we define the height of a DST and we prove that the memory complexity of the data structure needed to store a n-computer DST is $\mathcal{O}(\log(n))$ per computer.

Theorem 1: For every stage n, the number of computers of a node is bounded by a^{n-1} and b^n.

Proof: If is $\mid c_n \mid$ the number of computers of a stage n node, by recursion, we show that, for every stage n, with $1 \leq n \leq h$ where h is the DST height, $\mid c_n \mid$ is bounded by a^{n-1} and b^n.

If $n = 1 \wedge h = 1$, then $1 \leq \mid c_n \mid \leq b$ because of invariant 6.

If $n = 1 \wedge h \neq 1$, then $a \leq \mid c_n \mid \leq b$ because of invariant 6. So the theorem is true for $n = 1$.

We suppose that $a^n \leq \mid c_n \mid \leq b^n$ is true for a stage n with $n < h$. If $n+1 < h$, $a.a^n = a^{n+1} \leq \mid c_n \mid \leq b.b^n = b^{n+1}$ because of invariant 6.

If $n+1 = h$, $1.a^n = a^{(n+1)-1} \leq \mid c_n \mid \leq b.b^n = b^{n+1}$ because of invariant 6.

Then $\forall n, 1 \leq n \leq h, a^{n-1} \leq \mid c_n \mid \leq b^n$ ∎

Theorem 2 (DST height): If h is the height of a DST, then $\log_b(\mid \mathcal{C} \mid) \leq h \leq \log_a(\mid \mathcal{C} \mid) + 1$.

Proof: The set of computers $\mathcal{C}$ that constitutes a DST is also the set of computers that constitutes the root node.

From Theorem 1:

$$a^{h-1} \leq \mid \mathcal{C} \mid \Leftrightarrow h \leq \log_a(\mid \mathcal{C} \mid) + 1$$

and

$$\mid \mathcal{C} \mid \leq b^h \Leftrightarrow \log_b(\mid \mathcal{C} \mid) \leq h$$

Then, $\log_b(\mid \mathcal{C} \mid) \leq h \leq \log_a(\mid \mathcal{C} \mid) + 1$ ∎

Since the DST structure is defined we can present its use for traversal algorithms.

V. Traversal Algorithms

In this section, we describe two traversal algorithms. The first one intends to broadcast a message to every computer belonging to a DST. The second one is an optimized TTL-based search algorithm. More traversal algorithms can be found in [21].

A. Notations

The presented algorithms use the following notations:

- h is the height of a DST.
- `routing_table` is the routing table described in section IV-B.3. Every row is implemented by a list where the nth element is a reference to a computer of the child of index n. The list corresponding to the ith stage is stored in `routing_table`$[i]$.
- $c \rightarrow$`P`$(p_1, \ldots, p_n)$ is a call of the procedure `P` on the computer c with the parameters $p_1, \ldots, p_n$. To allow parallelism, these calls are asynchronous. The caller sends a message to the callee, instructing it of the call, and then executes the following instruction without waiting for the end of the call. When the procedure call is a remote procedure call written by the instruction $a \leftarrow (c \rightarrow$`F`$())$, a is set to the value returned by $c \rightarrow$`F`$()$. In this case, we suppose that there is a mechanism which automatically puts the return value of `F` in the variable a when the execution of the procedure `F` terminates.
- The variable *self* references the current computer.

B. Broadcast Algorithm

The aim of the broadcast algorithm is to efficiently send messages to every computer that is part of a DST. This algorithm is the simplest one and is very similar to the classical tree parallel traversal. The root node initiates the traversal by sending a message to all its children. Then, recursively, when a non-leaf node receives a message, it forwards it to its children.

Algorithm 1 Broadcast algorithm

```
    procedure Broadcast(msg)
        Broadcast_aux(h, msg)
    end procedure

    procedure Broadcast_aux(s, msg)
5:      if s = 0 then                ▷ The message is sent to a leaf
            process msg locally       ▷ End of the broadcast
        else                 ▷ The message is sent to a non-leaf node
            for all child ∈ routing_table[s] do
                child→Broadcast_aux(s − 1, msg)
10:         end for
        end if
    end procedure
```

The DST broadcast algorithm is presented by Algorithm 1. This algorithm uses two procedures. `Broadcast_aux` is a recursive procedure which broadcasts the message and `Broadcast` is the procedure which initializes the broadcast.

The procedure `Broadcast_aux` takes two parameters: msg the broadcasted message, and s the level of the called node in the tree (DST). Because non-leaf nodes are distributed over their descendants, every computer acts as leaf, as a node of *stage* 1, ... and as the root node. The parameter s tells the computer which node receives the message. If $s = h$, the computer must act as the root node. If $s = h - 1$, it must act as the child of the root node. If $s = 0$ (line 5), the computer must act as a leaf and it does not forward the message further.

If $s \neq 0$, the computer acts as a *stage* s forwarding node. So, this node forwards the message to its children. To do it, the computer takes the list of computers that represent the children of

its stage s node (line 8). For each of them, it asks him to forward the message (line 9) as a *stage* $s-1$ node, child of the stage s node.

The broadcast is initialized by the `Broadcast` procedure. This procedure must be called by a computer that is part of the DST. By calling `Broadcast_aux`(h, msg), it asks to himself to act as the root node[2] to forward the message msg.

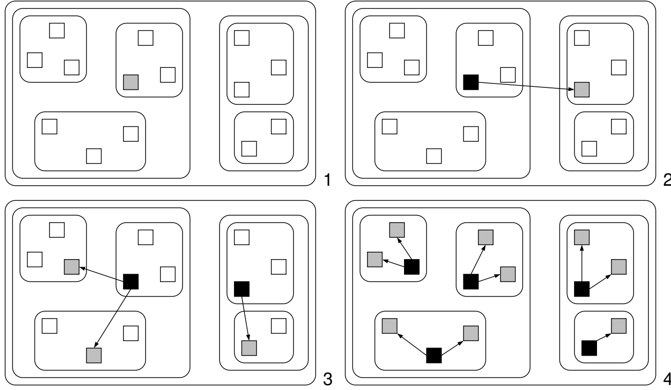


Fig. 11. Broadcast initiated by computer *e*.

The figure Fig. 11 is an example of a broadcast initiated by computer *e*. During the first step, *e* calls itself to broadcast the message. As the root node ϵ, *e* must forward the message to nodes 1 and 2 (see figure Fig. 10). Thus, during the second step, *e* sends a message to itself to contact node 1 and a message to *m* to contact node 2. As node 2, *m* must forward the message to node 21 and node 22. Thus, during the third step, *m* sends a message to itself to contact node 21 and a message to *o* to contact node 22.

Because a computer always uses itself as the representative of the nodes that contain it, every computer receives only one distant message. We can conclude that the number of distant messages of a broadcast on a n-node DST is $n-1$ because the computer that initializes a broadcast does not receive any distant messages. So, the complexity order of the broadcast is $\mathcal{O}(n)$ messages.

Traces of algorithm 1 show that the algorithm runs *h+1* recursions. So, the algorithm runs in $h+1 \leq \log_a(\mid \mathcal{C} \mid)+2$ steps. We can conclude that the algorithm time complexity is $\mathcal{O}(\log(n))$ time units.

C. Balancing of the Broadcast Algorithm

Traces of the broadcast algorithm illustrate well how the DST distributes the communication load between computers. The figure Fig. 11 is a trace of a broadcast initiated by *e*. The figure Fig. 12 is a trace of a broadcast initiated by *f*, another computer.

By comparing the traces of these two execution broadcasts we see that, depending of which computer initiates the broadcast, a computer can be a leaf or a non-leaf node which forwards the message. The figure Fig. 13 shows this by displaying the trace of two broadcasts — one initiated by *e* and the other initiated by *f* — on the same figure. Thus, contrary to classical broadcast trees, the DST distributes the load of forwarding messages between computers because every computer has the property to act as a leaf node and as a non-leaf node, depending on which computer the broadcast starts. This property makes every spanning tree different if the routing tables are different.

[2]because h is the height of the DST and *stage* n only contains the root node.

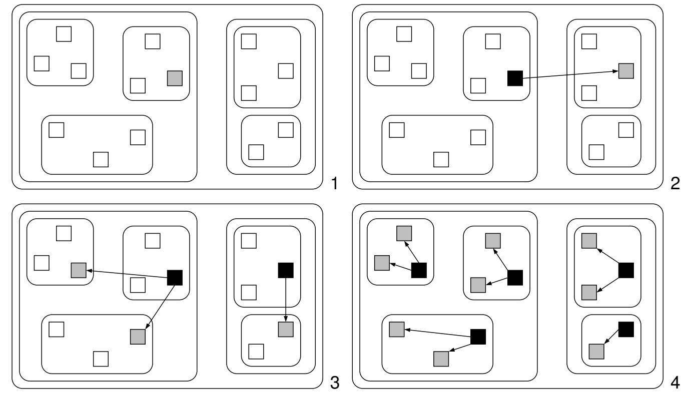


Fig. 12. Broadcast initiated by computer *f*.

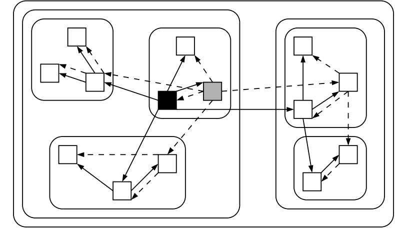


Fig. 13. Broadcast initiated by *e* and *f*.

D. Search Algorithm

The search algorithm aims at querying a number of computers which grows exponentially like a TTL-based graph flooding algorithm. This algorithm uses a broadcast like algorithm to query a sub-tree of *stage* 1, then a sub-tree of *stage* 2 and so on until the DST is completely flooded or until the query is positively answered.

The search algorithm uses two procedures. `Search_aux` is a recursive procedure which broadcasts requests in sub-trees and `Search` is the procedure which controls the search. To look for a resource matching a query, you need to call the `Search` procedure and pass your query as a parameter.

The `Search_aux` procedure takes two parameters to broadcast the request: s the height of the sub-tree and $query$ the searched resource description. If s equals 0, we query a leaf. In this case, the computer gathers the list of resources that match the query (line 20) and returns the list to the caller (line 31).

If s is not equal to 0, the computer broadcasts the query to its sub-tree of height $s-1$. To do it, for each child of its node of level s (line 23), the computer sends a message to its representative (line 8). This message asks the computer to gather the list of resources that match the query in its sub-tree of height $s-1$ (line 24). Then, the computer waits for the answers of the queried sub-trees and merges the lists of matching resources (lines 26–29). Finally, it returns the merged list to its caller (line 31).

The `Search` procedure takes one parameter ($query$) which describes the searched resource. The procedure starts by searching locally a resource that matches the query (line 2) and stores the

Algorithm 2 Search algorithm

procedure Search($query$)
  $l \leftarrow$Search_aux$(0, query)$
  $s \leftarrow 1$
  while $s \leq h \wedge l \equiv \emptyset$ **do**
5:   $tmp \leftarrow \emptyset$
    for all $child \in$ routing_table$[s]$ **do**
      if $s \neq self$ **then**
        $tmp[child] \leftarrow child \rightarrow$Search_aux$(s, query)$
      end if
10:   **end for**
    for all $ltmp \in tmp$ **do** ▷ Joins found resources
      $l \leftarrow l \cup ltmp$
    end for
    $s \leftarrow s + 1$
15: **end while**
  return l
end procedure

procedure Search_aux($s, query$)
  if $s = 0$ **then**
20:   $l \leftarrow$ List of local resources that match the query.
  else
    $tmp \leftarrow \emptyset$
    for all $child \in$ routing_table$[s]$ **do**
      $tmp[child] \leftarrow child \rightarrow$Search_aux$(s - 1, query)$
25:   **end for**
    $l \leftarrow \emptyset$ ▷ List of found resources
    for all $ltmp \in tmp$ **do** ▷ Join found resources
      $l \leftarrow l \cup ltmp$
    end for
30: **end if**
  return l
end procedure

list of found resources in the list l. Then, starting with a height of 1 (line 3), the procedure queries sub-trees of increasing height until querying the whole tree or until finding a resource (line 4).

To query a sub-tree of height s, the Search procedure sends a message (line 8) to a computer of each child of its node of level s (line 6) but one (line 7). We remind that a computer always chooses itself to represent nodes that contains the computer (invariant 11).

So, with the test $s \neq self$ (line 7), we avoid to query the sub-tree that contains the computer because this sub-tree is the sub-tree that has been queried during the previous iteration.

Once the queried subtrees return their lists of found resources, the computer merges the lists (lines 11–15) and prepares a new iteration in the case where no resources is found. At the end, it returns to the client the list of found resources that match the query (line 16).

The figure Fig. 14 is an example of a search initiated by the computer *e* where no resources is found. During the first step, the computer queries *e*, itself. Then, it queries its sub-tree of height 1 by querying *d* and *f*. During the third step, the computer queries its subtree of height 2 by broadcasting its search on node 11 and 13. Finally, it finishes to query the tree by broadcasting its search to node 2. Then, the whole DST is queried.

By following traces of the search algorithm, it is easy to notice that each computer is only queried once and that only $2.(n-1)$ messages are used to query the n computers of the DST. So, the complexity order of the search algorithm is $\mathcal{O}(n)$ messages. Because broadcasts are parallel, only $2s$ time units are needed to

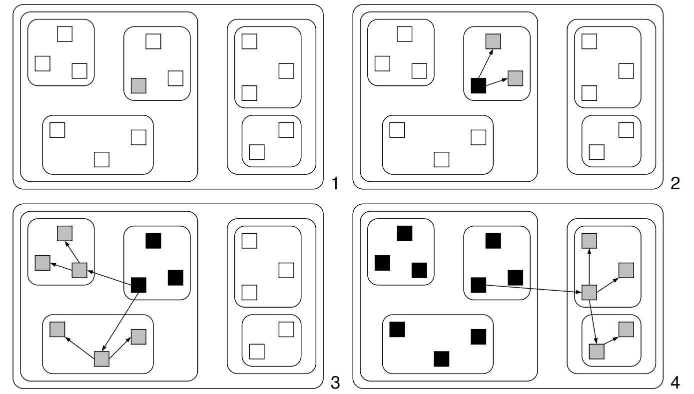


Fig. 14. Search initiated by computer *e*.

query a sub-tree of height s. Thus, to query a h stages DST, we need $2.(1+2+\ldots+h) = h(h+1)$ time units. We can conclude that the search algorithm has a time complexity order of $\mathcal{O}((\log(n))^2)$ time units.

VI. Distributing the DIET forwarding tree

In section III-A we explained that the DIET middleware uses a tree topology to query computational servers. To optimize searches, queries are only forwarded to sub-trees that have at least one computer which owns the software needed by the client. This is possible because every computational server informs its parent with the list of softwares that it proposes and every Local Agent informs its parent with the list of softwares proposed by its descendants. Fig. 15 displays this for two softwares a and b. This routing mechanism allows to send messages only to useful servers thus saving network resources. In this section, we explain how to do the same thing with a DST.

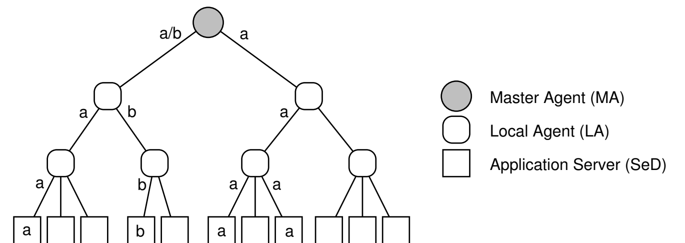


Fig. 15. A DIET search routing example.

When a computer e — which is always a leaf in a DST — installs a software, it informs its father node. Then, when the father node needs to forward a search query for this software, it knows that it has to forward the query to the computer e. Because, in a DST, a father node is distributed between several computers, the computer e has to forward the information on the software availability to all these computers by using the sub-tree rooted by its father.

In a more general way, when a node wants to provide a defined software, because the node is a computer or because one of its descendant wants to give access to a defined software, it must inform its father to forward it search queries that look for this software. To do this, it acts as its father node and broadcasts a message to all the descendants of its father node informing that it provides the software. A similar operation is done when a node stops to give access to a software.

This way, when a computer must act as a node X to forward a search query for an available server that proposes access to a software a, the computer knows which X's child must receive the query as this information has been forwarded to all computers that form the node X.

This routing mechanism is very useful when used with the search algorithm described in section V-D. The main issue with flooding algorithms is that they are inefficient to find rare resources because too much messages are generated. With this routing system we keep the characteristics of flooding algorithms but it is not longer needed to flood lots of computers to find a rare resource. We must note, in this case, that the trade-off is between the update process cost and the TTL value. The load is however balanced between the two processes.

We think that this approach is scalable because a software cannot be available on lots of computers and in few computers at the same time. In the DIET's context, servers are stable and do not come and go frequently. Thus, if a software is only present on few computers, the event that one of those computers comes or leaves is rare and the entire DST will be rarely completely flooded by the updating process. On the other hand, if lots of computers have the software, the event that one of those computers comes or leaves is common, but the message will be forwarded only to few computers because there is a high probability that its parent or grant-parent has another child that provides the same software.

This routing mechanism does not fit a context where there are lots of rare softwares that are proposed by only one computer and those computers come and go frequently because the DST will be completely flooded each time one of those computers comes or leaves. We can note however that there is, in our knowledge, no good solution to manage this kind of very dynamic environment.

VII. Construction Algorithms

Building a DST with centralized algorithms is straightforward. We quickly introduce them in the following section. But maintaining a DST with distributed algorithms is far more complex. A decentralized algorithm as been implemented in a simulator [22] and can be used as an example. Then we discuss the most interesting parts of this construction algorithm.

A. Centralized Algorithm

DST's construction algorithms must take care of two elements: making groups of nodes to build the hierarchy and linking computers together.

As described in section IV-A a non-leaf node is a complete graph of its children. Thus, we need an algorithm that puts nodes together to build parent nodes. As a reminder, nodes must have between a and b children, except the root and the leaves. If we choose a and b such as $b = 2a$, then the algorithm is very similar to B-tree construction algorithms [23]. This is interesting because B-tree building algorithms and their variants [24] are efficient and widely studied.

Once the algorithm finishes to build the hierarchy, it needs to connect computers together. Each computer is part of several nodes: its leaf node, the root node and several intermediate nodes. More precisely, every computer is a leaf node and also acts as all the parent node of its leaf. As explained in section IV-B every computer needs to know for each of its brother nodes (a brother node is an other child of its parent node) a computer that acts as this brother.

An simple method to do that is to run the following algorithm for every computer. The computer is a leaf. For every ancestor[3] of this leaf, take the list of its children. For every child, randomly take a computer that acts as this child and open a socket to this computer. This socket will be used as the link to this brother node.

Choosing randomly the computer that will be used to contact a brother is fine because the DST only specifies that the computer must be a descendant of the brother node. However, applications can use specialized algorithms to match their needs. For example, for fault robustness reasons, an application can take special care to avoid that every computer chooses the same computer to contact the brother node. Other applications could make other choices for load balancing reasons or depending on the physical network topology.

B. Distributed Algorithm

Using a centralized algorithm is easy to build a new DST each time you need it. But using a distributed algorithm is better to get an incremental approach, limit the number of exchanged messages and avoid the need of a global knowledge. This distributed algorithm is also mandatory in the case of self-organized peer networks.

Implementing the distributed algorithm is difficult for efficiency reasons and due to numerous details. However, we use some new techniques that greatly simplify the algorithm. In particular, we are able to use a simple and mutex-free election algorithm and a simple way to choose a computer that is a descendant of a defined node.

From the specification, a node cannot have more than b children. If several children can be added in parallel, then it is not trivial to avoid the case where several children are inserted when the parent node already has $b-1$ children and, thus, do not respect the specification. To avoid this problem, we implement a mutex that allows the insertion or the exclusion of only one child at a time, for a given node.

In our implementation, every node has a particular computer that acts as a mutex. Thus, before modifying the structure of the node, the algorithm acquires this mutex and releases it when the modification ends. Instead of implementing an election algorithm for choosing which computer serves as a mutex, we choose to use shared information for doing it.

When a new child is added, every computer that forms the parent node becomes aware of this new child. This is because if a computer does not know that it has a new child, the computer is not able to act as the parent node to forward messages to this child. If a computer knows who is the new child, it also knows who are the old children. Thus, every computer is able to maintain a list sorted by date where the oldest child is the first element and the youngest the last one. The advantage of this list is that all computers that form a node are able to share it without needing to exchange a message. In our implementation, this list used no memory (routing table is sorted instead of implementing a separated list) and uses few CPU to be maintained.

The mutex of a leaf's parent node is its oldest child. The mutex of a leaf's grant-parent node is the oldest child of its oldest child, and so on. By recursively taking the oldest child of a node, we end up taking a computer. This computer is used as the mutex of the node.

[3] parent node, grant-parent node, ..., root node.

VIII. Performances study

To prove the interest of the DST structure, we have developed a simulator. In this section, we present a study on the performances obtained with this simulator for the structure and the algorithms .

A. Description of the simulations

The aim of these simulations is to compare the behaviors and the performances of flooding search algorithms on top of three overlay network topologies: tree, pseudo-random graph and DST. Our comparison is limited to these two topologies because, as presented in section II, they are the most commonly used for self-organized networks whereas static or index oriented topologies cannot be used in this context.

Our first idea was to simulate and study the DST behavior on the Internet. However, experiences on Internet cannot be reproduced as the simulation conditions cannot be two times the same. Thus, we restrict ourselves to a model where all computers have one link, an access to every other computer through a central router and a limited bandwidth. This model is widely used to simulate overlay networks. It is not far from a realistic Internet model if we consider different bandwidth for different links and FIFO queues which guarantee that there is not two messages on a link at the same time. With this model, we can measure the impact of different factors, as the bandwidth or the physical network topology, on the execution performances.

Exact details of the simulations can be found by downloading the simulator [22]. The link model can be described the following way: a message[4] takes 1 ms to cross a link and there are FIFO queues assuring that there is no more than one message on a link at a given time. We perform simulations for populations of 10, 100, 1 000 and 10 000 peers to study the behavior of the algorithms when the overlay network scales up.

To simulate the execution of flooding algorithms, we use the algorithm described in [10] for the tree and the graph topologies: the initiator peer contacts its neighbors and waits for a reply; if no resource is found, then the initiator asks its neighbors to contact their neighbors and waits for their replies and so on until the end of the tree or the graph is reached. For the DST topology, we use the breadth first search like algorithm described in section V-D. It corresponds to an implementation of a flooding algorithm on this structure. Hundred different types of resources are available, and every computer has a probability of 10 % to own a resource of each type. Each search request stops either when it finds a node with the requested resources or when the whole structure is traversed.

About the overlay topologies characteristics, trees are bidirectional and their arity is 5. Graphs are also bidirectional, connected and the degree of each node is 5. Finally the DST is made in a way that each node has 5 children. These degrees were chosen because they show the best performances in our simulations. More precisely, we run some tests at various scales to find out these optimal degrees. Then, we use them for all the simulations by considering that these degrees are always optimal in our experiments[5]. However, these values depend on the links throughput and the probability to find a service. Changing one of these parameters implies that the chosen degrees would no longer be optimal.

[4]search messages are all about the same size, so they are modeled with the same length.

[5]This is always true when we check it. But systematic checking is not possible because of the amount of computing resources needed.

B. Search algorithm simulation

In this section, we discuss the results of the simulations done to evaluate the performances of the DST. For each simulated overlay network, we compare the performances of the three tested topologies. The two performance criteria studied here are the average time taken to process a search and its variation depending on the average load of the system. The system load is defined by the request arrival rate or frequency: the number of queries that are initiated per second for the whole system.

1) Overlay networks of 10 peers: The simulations results for the 10 peer overlay networks are displayed on the figure Fig. 16. The simulations show that the average time needed to process a request depends on the request arrival rate. This is an ordinary observation. When the number of initiated requests increases, the system becomes more and more loaded and messages spend more time in a waiting queue before being sent.

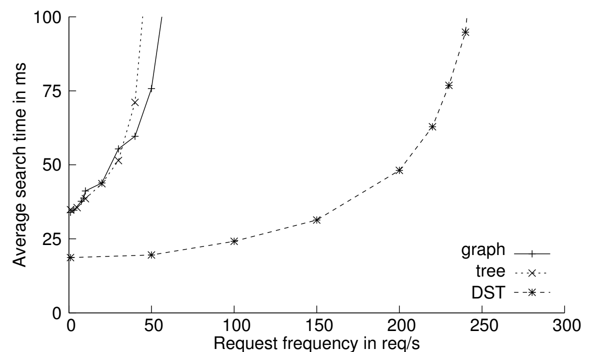


Fig. 16. Performances for networks of 10 peers

When the number of requests that enter the system becomes higher that the number of requests that leave it, the system becomes saturated. This saturation is easily identifiable for the DST on the figure Fig. 16: the average time needed to process a request increases slowly for frequencies from 1 req.s^{-1} to a frequency of 150 req.s^{-1}; but it increases very quickly for frequencies greater than 200 req.s^{-1}. On the other hand, the graph and the tree become very quickly saturated.

For a 10 peer overlay network, the simulations tell us that graphs and trees have similar performances. This is interesting because we expected to get better performances with the graph in term of time used to process a request when the system is not loaded (0.1 req.s^{-1}) as explained in section VIII-B.2. The more plausible explanation of this result is that some requests completely traverse the graph as no resource is found (we only have 10% of chance to find a resource on each peer and we only have 10 peers). So, these requests need an additional round to check that no other peer can be contacted. This is the final round where lots of messages are sent to traverse the latest links both ways while no untraversed peer remains. This additional round affects the global average search time.

Studies of individual load of peers also explain why the tree and the graph become saturated on a similar way: bottlenecks of both topologies have to support similar load. This can be explained by the fact that the tree has few bottlenecks that send an average of,

say $\bar{x}$ messages. With the graph traversal, which generates more messages, every peer sends an average of $\bar{x}$ messages. Because every peer has the same bandwidth, and because the topology performances are limited by their bottlenecks, both topologies become overloaded in a similar way at this scale.

The DST has the best behavior for these simulations. Because a DST is a tree, a search request only needs $2.n$ messages to query n peers, which is less than the graph. But, because it distributes the load of father nodes between its children, it does not suffer from the tree bottlenecks. Thus, the DST can support much more load than the other topologies at a scale of 10 peers. Note that the DST also generates a lower search time than the tree when the system is not loaded. The explanation is delivered in the next section which also discuses the results for a scale of 100 peers.

So the main result of this experience, except that the DST behaves as expected, is that for a small number of nodes or when the searched resource is rare the graphs topologies will generate an overhead to complete their traversal of links.

2) Overlay networks of 100 peers: The figure Fig. 17 presents the simulation results for 100 peers. Like before, the three topologies saturate when the query arrival rate becomes too high. From the simulations, it is clear that the tree has the worst performances in term of supported load. A frequency of 100 req.s^{-1} is enough to overload trees while graphs and DST start to be overloaded for a frequency of 700 req.s^{-1}. Before reaching this point, the average search time increases slightly with the average load of the system.

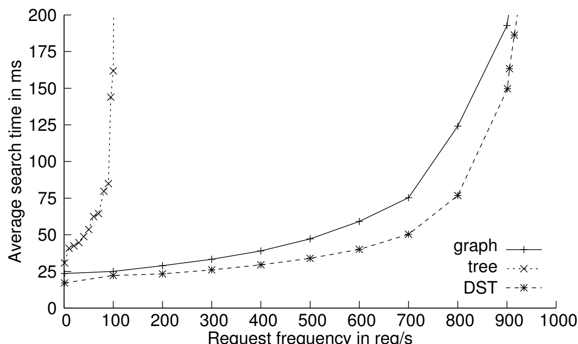


Fig. 17. Performances for networks of 100 peers

Graphs performances are roughly similar to the DST ones in term of supported load and average search time. To understand this similitude, it is important to remember that, in this experience, a query has 10 % chance to find the searched resource on each peer, that a search stops when at least one resource is found and that simulated graphs are not small world. For graphs of 100 peers and more, we notice that resources are found in one or two steps. Searches with three steps are rare and searches with more than three steps do not occur in graphs with at least 100 peers. A two steps search means that 31 peers[6] are traversed at most. From this value, we can conclude that the probability for a peer to be contacted at least twice is less than $\frac{1}{3}$. If every peer of a graph is contacted only once, then only $2.n$ messages are needed to query n peers and the average number of messages is optimal, like for the tree.

This is one of the results of these simulations: because each search needs to query a small part of the graph, few links are traversed to access already contacted peers. Thus, few peers are contacted twice by the same request and the number of exchanged messages is, with a big approximation, roughly similar to the tree's one. It depends on the number of nodes rather than on the number of links. For this reason, as a graph inherently distributes its load between its peers and as a DST is a tree that distributes the load of a father node between its descendants, the performances of both graph and DST are approximately similar in term of average search time and supported load. But this result is however only true when the searched resources have a high probability to be found. If they have not, then the graph topology will, as in the first experience, generates an overhead due to the traversal of unuseful links.

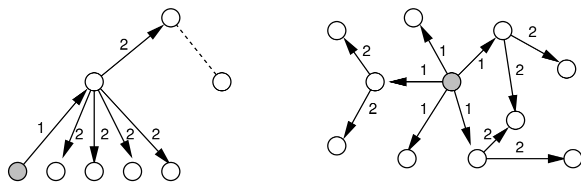


Fig. 18. The two first steps of a search in a tree and in a graph

We observe that tree topologies have a higher average search time than graphs even when the average load is very low. This is counterintuitive if we think that the average query time mainly depends on the number of exchanged messages and that the tree based traversal generates an optimal number of messages. But these results do not take into account the time taken for this traversal nor the position of the node into the tree. If we consider the root node, this traversal is optimal on a balanced tree even if the execution is done in parallel. But, if we consider a leaf of the same tree, the first step of a search only queries one peer, the father of the leaves. This behavior is illustrated by the figure Fig. 18 where the gray node only queries one node for the first round while the same step queries 5 peers in a graph or in a DST. At the second step the tree queries 5 peers when the graph or the DST can query 25 peers. Further, the tree behaves as if it always has one round late compared to the two other topologies, because tree leaves have one more hop to cross when they issue a search request. In a balanced tree with an arity of 5 and height n, there are more leaves (5^n) than non-leaf nodes ($\sum_{i=0}^{n-1} 5^i$), roughly a ratio of 5. Because every peer has the same probability to initiate a search, the majority of searches are initiated by leaf nodes. These searches are one round behind the other topologies and generate a higher latency. This fact explains why the average search time of the tree is more important than the two others even when the system is under light load.

So the main result of this experience is that the tree topologies are not efficient for the leaves, compared to graphs or DST, as their first request will only query one node: their parent.

3) Large overlay networks: The figure Fig. 19 gives the simulated performances for overlay networks of 1 000 peers. A load of 300 req.s^{-1} is enough to saturate the tree. Graphs and DST start to be overloaded around 8 000 req.s^{-1}. Before being overloaded, the average search time of DST and graphs increases slowly when the load is increasing.

Graph's performances are close to the DST's ones. The previous observation, which claims that graphs roughly send $2.n$ messages to query n peers, is more pertinent in this case because peers have a very low probability to be queried twice due to the total number of peers as explained for the previous experience.

[6] $31 = 5^0 + 5^1 + 5^2$.

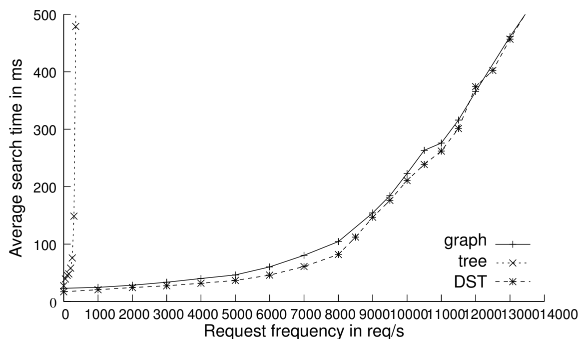


Fig. 19. Performances for networks of 1 000 peers

Simulation results for topologies of 10 000 peers (see figure Fig. 20) do not show anything new. Trees start to be overloaded around 1 000 req.s^{-1} and the two other topologies support at least a load of 40 000 req.s^{-1}. The average search time of graphs and DST increases slowly and linearly as a function of the load.

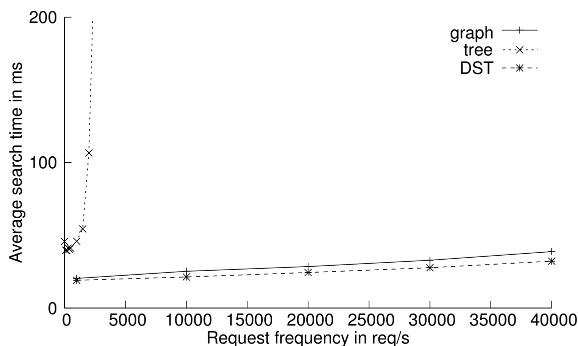


Fig. 20. Performances for networks of 10 000 peers

We did no simulation with more that 40 000 req.s^{-1} due to the amount of needed RAM. Every request sends several messages and every message uses some memory space until it arrives to its destination, then this memory is freed. A simulation with a load of more that 40 000 req.s^{-1} is enough to consume 1 GB of memory. With bigger simulations our system swaps and the average CPU usage falls to 3 %. This is unacceptable knowing that the simulation of 40 000 req.s^{-1} takes between 2 and 3 days of computation with a CPU usage of 100%.

4) Scalability of the three topologies: Until now, we have discussed the performances for each scale separately. Here we present how the number of peers affects the performances of each topology.

Adding new peers allows to increase the performances in term of supported load. 40 req.s^{-1} saturate a tree of 10 peers, 100 req.s^{-1} saturate a tree of 100 peers, 300 req.s^{-1} saturate a tree of 1 000 peers and 1 000 req.s^{-1} saturate a tree of 10 000 peers. This is not very efficient because multiplying the number of peers by 100 (from 100 peers to 10 000 peers) only multiplies the supported load by 10 (from 100 req.s^{-1} to 1 000 req.s^{-1}).

The DST has a good scalability. The supported load increases linearly with the number of peers: 150 req.s^{-1} for 10 peers, 700 req.s^{-1} for 100 peers, 8 000 req.s^{-1} for 1 000 peers and more than 40 000 req.s^{-1} for 10 000 peers. When the system is not saturated, the average search time is stable: it varies from 25 ms to 75 ms whatever the number of peers.

A performance of 150 req.s^{-1} on 10 peers (15 per peer) for a DST may look inconsistent with the performance of 700 req.s^{-1} for 100 peers (7 per peer). But, the reason is that a two step search only contacts 10 peers with only 10 peers, while the same search contacts 25 peers in a DST of 100 peers. Less peers contacted means that less messages are generated and so implies better supported load.

Globally on the previous figures (10 to 10 000 nodes), the scalability of the graph is similar to the scalability of the DST except for small scales where the graph get bad performances. This similarity comes from the fact that both topologies behave in a similar way when the probability to find a resource on each peer is fixed (10 % in our simulation) and that the number of peers is high enough.

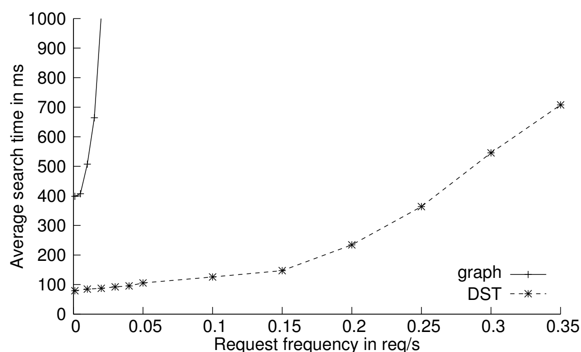


Fig. 21. Performances for networks of 1 000 peers when no resources are available

However, if the probability to find a resource is low enough that the request is more widely spread and that peers receive a query several times, then the performances of graphs cannot cope with the performances of DST. The figure Fig. 21 presents the performances of graphs and DST of 1 000 computers when the probability to find a resource is 0 %. In this case, the DST's performances are better than the graph's ones for two reasons:

1) a DST sends fewer messages as it uses the spanning tree for its traversal algorithm. The number of sent messages depends on the number of nodes while it depends on the number of links for the graph;
2) a DST distributes fairly its load between computers as the spanning trees used by the nodes are distributed across the network nodes. Thus no bottleneck is generated compared to the tree topology.

Note that a frequency of 1 broadcast request every 5 seconds is enough to saturate a DST of 1 000 computers. This is normal as expanding ring algorithms, which generates several waves of search, are not efficient for this kind of applications. A DST is more designed to support search requests that are just partially spread on the structure.

IX. CONCLUSION

In this paper we have presented the Distributed Spanning Tree structure: an interconnection proposition which aims at connecting distributed nodes of a self-organized overlay network. This structure is designed to support scalable searches and traversal algorithms. It provides characteristics similar to trees while avoiding the bottlenecks generated by the tree root. It behaves better than randomly generated graphs as it only needs

two times the number of nodes messages to query all nodes and, thus, significantly improves traversal algorithm efficiency. Simulations validate that the DST improves the performances of search algorithms, whatever the size of the network, from ten to several thousands nodes. The DST provides also good results when broadcasting a request on the whole network. The main drawback of this structure is its cost of construction but this cost is spread out over the life time of the DST as the structure is incrementally generated by nodes that join (or leave) the overlay network.

Future works include the detailed study of the DST properties. In particular, we are interested in tuning the value of the two constants a and b (respectively min and max size of the groups) to guarantee the stability of the DST. From these values will depend the number and the frequency of group splitting and merging. If these two values are too distant, the balance of the DST will not be as good and this will probably impact the algorithm efficiency.

In the design of the DST, we assume that the global number of nodes does not vary too much on a short period. So, an other key point is to show that the behavior remains stable in these conditions and to study its evolution if this assumption is not verified anymore. We plan to evaluate the maintenance cost when the number of nodes varies more than expected and to find the threshold above which the maintenance becomes too costly.

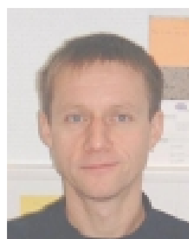
Acknowledgment

This work was supported by the INRIA GRAAL project and by the Région of Franche-Comté.

References

[1] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lims, “A survey and comparison of peer-to-peer overlay network schemes,” *IEEE Communications Survey and Tutorial*, vol. 7, no. 2, April 2005.
[2] S. Campbell, M. Kumar, and S. Olariu, “The hierarchical cliques interconnection network,” *J. of Parallel and Distributed Computing - Elsevier*, vol. 64, pp. 16–28, 2004.
[3] Q. M. Malluhi and M. A. Bayoumi, “The hierarchical hypercube: A new interconnection topology for massively parallel systems,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, no. 1, pp. 17–30, Jan. 1994.
[4] S. Gribble, E. Brewer, J. Hellerstein, and D. Culler, “Scalable, distributed data structures for internet service construction,” in *Symp. OSDI*, 2000.
[5] A. Rowstron and P. Druschel, “Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems,” *LNCS*, vol. 2218, pp. 329–350, 2001.
[6] C. G. Plaxton, R. Rajaraman, and A. W. Richa, “Accessing nearby copies of replicated objects in a distributed environment,” in *ACM Symp. on Parallel Algo. and Arch.*, 1997, pp. 311–320.
[7] Clip2. (2001) The gnutella protocol specification v0.4. [Online]. Available: www9.limewire.com/developer/gnutella_protocol_0.4.pdf
[8] S. Networks. (2004) The glossary: Supernodes. [Online]. Available: http://www.kazaa.com/us/help/glossary/supernodes.htm
[9] *JXTA v2.0 Protocols Specification*, Sun Microsystems Specification, 2004. [Online]. Available: spec.jxta.org/nonav/v1.0/docbook/JXTAProtocols.html
[10] S. Dahan, J. Nicod, and L. Philippe, “Scalability in a GRID server discovery mechanism,” in *10th IEEE Int. Workshop on Future Trends of Distributed Computing Systems, FTDCS 2004*. Suzhou, China: IEEE Press, May 2004, pp. 46–51.
[11] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, “SplitStream: High-bandwidth content distribution in cooperative environments,” in *Proc. of the 2nd International Workshop on Peer-to-peer System*, Berkeley, CA, Feb. 2003.
[12] B. Cohen. (2003) Incentives build robustness in bittorrent. [Online]. Available: http://bitconjurer.org/BitTorrent/bittorrentecon.pdf
[13] M. Izal, G. Urvoy-Keller, E. Biersack, P. Felber, A. Al Hamra, and L. Garcés-Erice, “Dissecting bittorrent: Five months in a torrent’s lifetime,” in *Passive and Act. Net. Meas., 5th In. Wk.*, ser. LNCS, vol. 3015, Antibes Juan-les-Pins, France, Apr. 2004, pp. 1–11.
[14] E. Caron and F. Desprez, “Diet: A scalable toolbox to build network enabled servers on the grid,” *International Journal of High Performance Computing Applications*, vol. 20, no. 3, pp. 335–352, 2006.
[15] E. Caron, F. Desprez, F. Lombard, J.-M. Nicod, L. Philippe, M. Quinson, and F. Suter, “A scalable approach to network enabled servers,” in *Proceedings of the 8th International EuroPar Conference*, ser. Lecture Notes in Computer Science, B. Monien and R. Feldmann, Eds., vol. 2400. Paderborn, Germany: Springer-Verlag, Aug. 2002, pp. 907–910.
[16] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova, “A GridRPC model and API for end-user applications,” Global Grid Forum, Tech. Rep., Sep. 2004.
[17] S. Contassot-Vivier, F. Lombard, J.-M. Nicod, and L. Philippe, “Evaluation of the DIET hierarchical metacomputing architecture,” *Parallel and Distributed Computing Practices. Special Issue on Parallel Numeric Algorithms on Faster Computers*, vol. 5, no. 4, pp. 64–76, Dec. 2002.
[18] Z. Tari and G. Craske, “A query propagation approach to improve CORBA trading service scalability,” in *International Conference on Distributed Computing Systems*, 2000, pp. 504–511.
[19] E. Anceaume, A. K. Datta, M. Gradinariu, G. Simon, and V. A., “DPS: Self-* dynamic reliable content-based publish/subscribe system,” IRISA, Tech. Rep. 1665, Dec. 2004.
[20] E. Anceaume, M. Gradinariu, A. K. Datta, G. Simon, and A. Virgillito, “A semantic overlay for self-* peer-to-peer publish/subscribe,” in *26th IEEE International Conference on Distributed Computing Systems*, Lisboa, Portugal, Jul. 2006, pp. 22–30.
[21] S. Dahan, “Distributed Spanning Tree algorithms for large scale traversals,” in *Procedings of the11th International Conference on Parallel and Distributed Systems, ICPADS 2005*, vol. 1. Fukuoka, Japan: IEEE Computer Society, Jul. 2005, pp. 453–459.
[22] Simulator, “graal.ens-lyon.fr/~jmnicod/simulator-12-06.tar.gz,” 2006.
[23] S. Dahan, J. Nicod, and L. Philippe, “The Distributed Spanning Tree: A scalable interconnection topology for efficient and equitable traversal,” in *5th Int. Symposium on Cluster Computing and the Grid*. Cardiff, UK: IEEE Press, May 2005.
[24] 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.

Sylvain Dahan obtained his Ph.D in 2005 from Université of Franche-Comté, France. From 2006 to 2008 he was an Associate Researcher in the High Performance Computing System Laboratory of the University of Tsukuba in Japan. He is currently engineer at Altran consulting.



Laurent Philippe obtained his Ph.D degree in 1993 at the Université of Franche-Comté and his HDR in 2000. He is currently a full Professor in Computer Science Laboratory at the Universit of Franche-Comté (LIFC). His main research interests include distributed systems, middleware, scheduling algorithms for distributed heterogeneous platforms.

Jean-Marc Nicod obtained his Ph.D degree in 1997 at the École Normale Supérieure of Lyon (ENS) and his HDR at the Université of Franche-Comté in 2006. He is currently an Assistant Professor in the Computer Science Laboratory at the Universit of Franche-Comté (LIFC). His research interests include parallel programming, distributed systems, algorithms and scheduling for distributed heterogeneous systems.