Snap-stabilizing Prefix Tree for Peer-to-peer Systems

Eddy Caron[1], Frédéric Desprez[1], Franck Petit[2], and Cédric Tedeschi[1]

[1] LIP Laboratory
UMR CNRS-ENS Lyon-UCB Lyon-INRIA 5668
46 Allée d'Italie, 69364 Lyon Cedex 07, France
[2] LaRIA Laboratory
CNRS-University of Picardie
5, rue du Moulin Neuf, 80000 Amiens, France

Abstract. *Resource Discovery* is a crucial issue in the deployment of computational grids over large scale peer-to-peer platforms. Because they efficiently allow range queries, *Tries* (*a.k.a., Prefix Trees*) appear to be among promising ways in the design of distributed data structures indexing resources. *Self-stabilization* is an efficient approach in the design of reliable solutions for dynamic systems. A *snap-stabilizing* algorithm guarantees that it always behaves according to its specification. In other words, a snap-stabilizing algorithm is also a self-stabilizing algorithm which stabilizes in 0 steps.
In this paper, we provide the first snap-stabilizing protocol for trie construction. We design particular tries called *Proper Greatest Common Prefix* (PGCP) Tree. The proposed algorithm arranges the n label values stored in the tree, in average, in $O(h+h')$ rounds, where h and h' are the initial and final heights of the tree, respectively. In the worst case, the algorithm requires an $O(n)$ extra space on each node, $O(n)$ rounds and $O(n^2)$ actions. However, simulations show that, using relevant data sets, this worst case is far from being reached and confirm the average complexities, making this algorithm efficient in practice.

Keywords: Peer-to-peer systems, Fault-tolerance, Self-stabilization, Snap-stabilization, Grid computing.

1 Introduction

These last few years have seen the development of large scale grids connecting distributed resources (computation resources, storage facilities, computation libraries, etc.) in a seamless way. This is now an efficient alternative to supercomputers for solving large problems such as high energy physics, bioinformatics or simulation. However, existing middleware systems always require a minimal stable centralized infrastructure and are not usable over dynamic large scale distributed platforms. To cope with the characteristics of these future platforms, it has been widely suggested to use peer-to-peer technologies inside middleware [22]. Early distributed hash tables (DHT), designed for very large scale platforms, *e.g.,* to share files over the Internet, have several major drawbacks. Among them, there is the fact that they only support exact match queries. An important amount of work has recently been undertaken to allow more complex querying over peer-to-peer systems. A promising way to achieve this is the use of *tries* (*a.k.a., prefix trees*). Trie-based approaches outperform other ones by efficiently supporting range queries and easily extending to multi-criteria searches.

Unfortunately, although fault tolerance is a mandatory feature of systems aiming at being deployed at large scale (to avoid data loss and allow a correct routing of messages through the network), tries only offer a poor robustness in dynamic environment. The crash of one or several nodes leads to the loss of stored objects and to the split of the trie into several subtries. These subtries may then not reconnect correctly, making the trie invalid and thus unable to process queries. Among recent trie-based approaches, the fault-tolerance is either ignored, or handled by preventive mechanisms, intensively using replication which can be very costly in terms of computing and storage resources. Afterward, the purpose is to compute the right trade-off between the replication cost and the robustness of the system. Nevertheless, replication does not formally ensure the recovery of the system after arbitrary failures. From this point on, it remains only to use a strategy based on the *best-effort* approach. This is why we believe that such systems could take advantage of using self-stabilization techniques in order to satisfy the fault tolerance requirements.

The concept of self-stabilization [16] is a general technique to design a system tolerating arbitrary transient faults. A self-stabilizing system, regardless of the initial states of the processors and initial messages in the links, is guaranteed to converge to the intended behavior in finite time. Thus, a self-stabilizing system does not need to be reinitialized and is able to recover from transient failures by itself.

In this paper, we propose a snap-stabilizing distributed algorithm to build a *Proper Greatest Proper Common Prefix* (GPCP) Tree starting from any labeled rooted tree. A *snap-stabilizing* [13] algorithm ensures that the system always maintains the desirable behavior and is obviously optimal in stabilization time. The property of snap-stabilization is achieved within the well-known Dijkstra's theoritical model [15] where in each computation step, each node can atomically read variables (or, registers) owned by its neighboring nodes.

The proposed algorithm arranges the n label values (each node holds a single label) stored in the tree, in average, in $O(h + h')$ rounds, where h and h' are the initial (before reconstruction) and final (after reconstruction) height of the tree, respectively. In the worst case, the algorithm requires an $O(n)$ extra space on a given node, $O(n)$ rounds and $O(n^2)$ operations. However, simulations show that, using relevant data sets, the worst case is far from being reached and confirm the average complexity. It also shows the practical efficiency of the proposed algorithm and the benefit of snap-stabilization in the design of efficient algorithms for unreliable, dynamic environments where the best-effort seems to be a valuable strategy.

In Section 2, we summarize recent peer-to-peer technologies used for resource discovery and their fault-tolerance mechanisms, followed by similar works undertaken in the field of self-stabilization. In Section 3, we describe the abstract model in which our algorithm is designed, and present what it means for a distributed algorithm to be snap-stabilizing. We also specify the PGCP Tree and related distributed data structures. In Section 4, the snap-stabilizing scheme protocol is presented, and its correctness proof and complexities discussed. Simulation process are explained and results given in Section 5. Finally, we conclude by summarizing the contribution of the paper and a brief description of next steps in this work.

2 Related Work

First peer-to-peer algorithms aiming at retrieving objects were based on the flooding of the network, overloading the network while providing non-exhaustive responses. Addressing both the scalability and the exhaustiveness issues, the distributed hash tables [25, 26, 29], logical hops required to route and the local state grow logarithmically with the number of nodes participating in the system. Unfortunately, DHTs present several major drawbacks. Among them, the rigidity of the requesting mechanism, only allowing exact match queries, hinders its use over distributed computational platforms that require more complex meanings of search.

A large amount of work tackles the opportunity to allow more flexibility in the retrieval process over structured peer-to-peer networks. Peer-to-peer systems users have been given the opportunity to plug different technologies on DHTs, such as the ability to retrieve resources described by semi-structured languages [5], to manage data thanks to traditional database operations [30], or to support multi-attribute range queries [1, 23, 27, 28]. Among this last series of work supporting multi-attribute range queries, a new kind of overlay, based on tries, has emerged. Trie-structured approaches outperform others in the sense that logarithmic (or constant if we assume an upper bound on the depth of the trie) latency is achieved by parallelizing the resolution of the query in the several branches of the trie.

Prefix Hash Tree (PHT) [24] dynamically builds a trie of the given key-space as an upper layer and maps it over any DHT-like network. Obviously, the architecture of PHT results in the multiplication of the complexities of the trie and of the underlying DHT. The problem of fault tolerance is then delegated to the DHT layer. Skip Graphs, introduced in [3], are also similar to a trie, but rely on skip lists, allowing the use of their probabilistic fault tolerance. Nevertheless, a repair mechanism of the particular skip graph structure is provided. Nodewiz [6], another trie-structured overlay does not address the fault-tolerance problem by assuming the nodes reliable. Finally, P-Grid [14] tolerance is based on probabilistic replication. Initially designed for the purpose of service discovery over dynamic computational grids and aimed at solving some drawbacks of these previous approaches, we recently developed a novel architecture, based on a logical greatest common prefix tree [11]. This structure, more formally described in the following, is dynamically built as objects, *e.g.,* computational services, are declared by some servers. The fault tolerance is also addressed by replication of nodes and links of the tree. Another advantage of the technology presented in [11] is its ability to greedily take into account the heterogeneity of the underlying physical network to make a more efficient tree overlay.

To summarize, the fault-tolerance issue is mostly either ignored, delegated or replication-based. In [10], we provided a first alternative to the replication approach. The idea was to let the trie crash and to *a posteriori* reconnect and reorder the nodes. However, this protocol assumed the validity of subtries being reordered, thus limiting the field of initial configurations being handled and repaired. In the following sections, we present a new protocol able to repair any labeled rooted tree to make a valid greatest common prefix tree and thus to offer a general systematic mechanism to maintain distributed tries.

In the self-stabilizing area, some investigations take interest in maintaining distributed data structures. The solutions in [19–21] focus on binary heap and 2-3 trees. Several approaches have also been considered for a distributed spanning tree maintenance *e.g.,* [2, 4, 12, 17, 18]. In [18], a new model for distributed algorithms designed for large scale systems is introducted. In [7], the authors presented the first snap-stabilizing distributed solution for the Binary Search Tree (BST) problem. Their solution requires $O(n)$ rounds to build the BST, which is proved to be asymptotically optimal for this problem in the same paper.

3 Preliminaries

In this section, we first present the distributed system model used in the design of our algorithm. Then, we recall the concept of snap-stabilization and specify the distributed data structures considered.

3.1 Distributed System

The distributed algorithm presented in this paper is intended for practical *peer-to-peer* (P2P) networks. A P2P network consists of a set of asynchronous *physical* nodes with distinct IDs, communicating by message passing. Any physical node P_1 can communicate with any physical node P_2, provided P_1 knows the ID of P_2 (ignoring physical routing details). Each *physical* node maintains one or more *logical* nodes of the distributed *logical* tree. Our algorithm is run inside all these *logical* nodes. Note that the tree topology is susceptible to changes during its reconstruction. Each *logical* node of the tree has to be considered mapped on a *physical* node of the underlying network. However, the mapping process falls beyond the scope of this paper.

In order to simplify the design, proofs, and complexity analysis of our algorithm, we use the theoretical formal *state model* introduced in [15]. We apply this model on logical nodes (or simply, nodes) only. The message exchanges are modeled by the ability of a node to read the variables of some other nodes, henceforth referred to as its neighbors. A node can only write to its own variables. Each action is of the following form: $<$ *label* $>::$ $<$ *guard* $>$ $\rightarrow$ $<$ *statement* $>$. The guard of an action in the program of p is a boolean expression involving the variables of p and its neighbors. The statement of an action of p updates one or more variables of p. An action can be executed only if its guard evaluates to true. We assume that the actions are atomically executed, meaning the evaluation of a guard and the execution of the corresponding statement of an action, if executed, are done in one atomic step.

The *state* of a node is defined by the values of its variables. The *state* of a system is a product of the states of all nodes. In the sequel, we refer to the state of a node and the system as a *state* and a *configuration*, respectively. Let a relation denoted by $\mapsto$, on $\mathcal{C}$ (the set of all possible configurations of the system). A *computation* of a protocol $\mathcal{P}$ is a *maximal* sequence of configurations $e = (\gamma_0, \gamma_1, ..., \gamma_i, \gamma_{i+1}, ...)$, such that for $i \geq 0, \gamma_i \mapsto \gamma_{i+1}$ (a single *computation step*) if γ_{i+1} exists, or γ_i is a terminal configuration.

A processor p is said to be *enabled* in γ ($\gamma \in \mathcal{C}$) if there exists at least an action A such that the guard of A is true in γ. We consider that any enabled node p is *neutralized* in the computation step $\gamma_i \mapsto \gamma_{i+1}$ if p is enabled in γ_i and not enabled in γ_{i+1}, but does not execute any action between these two configurations (the neutralization of a node represents the following situation: At least one neighbor of p changes its state between γ_i and γ_{i+1}, and this change effectively made the guard of all actions of p false.) We assume an *unfair and distributed daemon*. The *unfairness* means that even if a processor p is continuously enabled, then p may never be chosen by the daemon unless p is the only enabled node. The *distributed* daemon implies that during a computation step, if one or more nodes are enabled, then the daemon chooses at least one (possibly more) of these enabled nodes to execute an action.

In order to compute the time complexity, we use the definition of *round*. This definition captures the execution rate of the slowest node in any computation. The set of all possible computations of $\mathcal{P}$ is denoted as $\mathcal{E}$. The set of possible computations of $\mathcal{P}$ starting with a given configuration $\alpha \in \mathcal{C}$ is denoted as $\mathcal{E}_\alpha$. Given a computation e ($e \in \mathcal{E}$), the *first round* of e (let us call it e') is the minimal prefix of e containing the execution of one action of the protocol or the neutralization of every enabled node from the first configuration. Let e'' be the suffix of e, *i.e.,* $e = e'e''$. Then *second round* of e is the first round of e'', and so on.

3.2 Snap-Stabilization

Let $\mathcal{X}$ be a set. $x \vdash P$ means that an element $x \in \mathcal{X}$ satisfies the predicate P defined on the set $\mathcal{X}$.

Definition 1 (Snap-stabilization). *The protocol $\mathcal{P}$ is snap-stabilizing for the specification $\mathcal{SP}_\mathcal{P}$ on $\mathcal{E}$ if and only if the following condition holds:* $\forall \alpha \in \mathcal{C} : \forall e \in \mathcal{E}_\alpha :: \; e \vdash \mathcal{SP}_\mathcal{P}$.

3.3 Proper Greatest Common Prefix Tree

Let an ordered alphabet A be a finite set of letters. Denote $\prec$ an order on A. A non empty *word* w over A is a finite sequence of letters $a_1, \ldots, a_i, \ldots, a_l$, $l > 0$. The *concatenation* of two words u and v, denoted $u \circ v$ or simply uv, is equal to the word $a_1, \ldots, a_i, \ldots, a_k, b_1, \ldots, b_j, \ldots, b_l$ such that $u = a_1, \ldots, a_i, \ldots, a_k$ and $v = b_1, \ldots, b_j, \ldots, b_l$. Let ϵ be the *empty word* such that for every word w, $w\epsilon = \epsilon w = w$. The *length* of a word w, denoted by $|w|$, is equal to the number of letters of w—$|\epsilon| = 0$.

A word u is a *prefix* (respectively, *proper prefix*) of a word v if there exists a word w such that $v = uw$ (resp., $v = uw$ and $u \neq v$). The *Greatest Common Prefix* (resp., *Proper Greatest Common Prefix*) of a collection of words $w_1, w_2, \ldots, w_i, \ldots$ ($i \geq 2$), denoted $GCP(w_1, w_2, \ldots, w_i, \ldots)$ (resp. $PGCP(w_1, w_2, \ldots, w_i, \ldots)$), is the longest prefix u shared by all of them (resp., such that $\forall i \geq 1, u \neq w_i$).

Definition 2 (PGCP Tree). *A* Proper Greatest Common Prefix Tree *is a labeled rooted tree such that each node label is the Proper Greatest Common Prefix of every pair of its children labels.*

In the design of our protocol, we also needs the relaxed form of PGCP Tree defined as follows:

Definition 3 (PrefixHeap). *A PrefixHeap is a labeled rooted tree such that each node label is the Proper Greatest Common Prefix of all its children labels.*

4 Snap-stabilizing PGCP Tree

In this section, we present the snap-stabilizing PGCP tree maintenance. We provide a detailed explanation of how the algorithm works from initialization until the labels are arranged in the tree such that it becomes a PGCP tree. Next, the proof of snap-stabilization and complexity issues are given.

4.1 The algorithm

The code of our solution is shown in Algorithms 1 and 2. We assume that initially, there exists a labeled rooted tree spanning the network. Every node p maintains a finite set of children $C_p = \{c_1, \ldots, c_k\}$, which contains the addresses of its children in the tree. Each node p is able to know the address of its father using the macro f_p. The uniqueness of the father is ensured by the use of the function $MinID(S)$ which returns the minimal values in the set S[3]. So, each node p can locally determine if it is either (1) the single *root* of the spanning tree (f_p is unspecified), (2) an *internal* node (f_p is specified and $C_p \neq \emptyset$), or (3) a *leaf* node ($c_p = \emptyset$). In the sequel, we denote the set of nodes in the tree rooted at p

[3] In a real P2P network, the relationship child/father is easily preserved by exchanging messages between a child node and its father.

as T_p (hereafter, also called the *tree* T_p) and the *height* of the tree rooted at p as $h(T_p)$.

Each node p holds a label l_p and a state S_p in $\{I, B, H\}$[4]—stand for *Idle*, *Broadcast*, and *Heapify*, respectively. The algorithm uses two basic functions to create and delete nodes from the tree. The **NEWNODE**($lbl, st, chldn$) function creates a new node labeled by lbl, whose initial state is st and with a set of children initialized with $chldn$. Once the new node created by this function is integrated to a set of children, the f_p macro will ensure its father to be correctly set. Finally, the same f_p macro will set the father variable of nodes in $chldn$. The **DESTROY**(p) function is called to stop the process of a given node, (its reference should have been previously deleted from any other node).

Algorithm 1 Snap-Stabilizing PGCP Tree — Variables, Macros, and Actions.

```
Variables:   $l_p$, the label of $p$
             $C_p = \{c_1, \ldots, c_k\}$
             $S_p = \{I, B\}$ if $p$ is the root, $\{I, H\}$ if $p$ is a leaf node, $\{I, B, H\}$ otherwise ($p$ is an internal node)

Macros:      $f_p \equiv MinID(\{q :\ p \in C_q\})$
             $SameLabel_p(L) \equiv \{c \in C_p \mid (l_c = L)\}$
             $SameGCP_p(L) \equiv \{c_1, c_2, \ldots, c_k \in C_p \mid GCP(c_1, c_2, \ldots, c_k) = L\}$
             $SamePGCP_p(L) \equiv SameGCP_p(L) \setminus \{c \in SameGCP_p(L) \mid l_c = L\}$

Actions:
                                          {For the root node}
      InitBroadcast ::   $S_p = I \wedge (\forall c \in C_p \mid S_c = I) \longrightarrow S_p := B;$
         InitRepair ::   $S_p = B \wedge (\forall c \in C_p \mid S_c = H) \longrightarrow$ HEAPIFY();REPAIR(); $S_p := I;$
                                          {For the internal nodes}
   ForwardBroadcast ::   $S_p = I \wedge S_{f_p} = B \wedge (\forall c \in C_p \mid S_c = I) \longrightarrow S_p := B;$
      BackwardHeap ::    $S_p = B \wedge S_{f_p} = B \wedge (\forall c \in C_p \mid S_c = H) \longrightarrow$ HEAPIFY(); $S_p := H;$
     ForwardRepair ::    $S_p = H \wedge S_{f_p} = I \wedge (\forall c \in C_p \mid S_c \in \{H, I\}) \longrightarrow$ REPAIR(); $S_p := I;$
    ErrorCorrection ::   $S_p = B \wedge S_{f_p} \in \{H, I\} \longrightarrow S_p := I;$
                                          {For the leaf nodes}
           InitHeap ::   $S_p = I \wedge S_{f_p} = B \longrightarrow S_p := H$
          EndRepair ::   $S_p = H \wedge S_{f_p} = I \longrightarrow S_p := I;$
```

The basic idea of the algorithm is derived from the fast version of the snap-stabilizing PIF in [8] and runs in three phases: The root initiates the first phase, called the *Broadcast* phase, by executing Action *InitBroadcast*. All the internal nodes in the tree participate in this phase by forwarding the broadcast message to their descendants — Action *ForwardBoradcast*. Once the broadcast phase reaches the leaves, they initiates the second phase of our scheme, called the *heapify* phase, by executing Action *InitHeap*.

During the heapify phase, a *PrefixHeap* is built — refer to Definition 3. We also ensure in this phase that for every node p, p is a single node in T_p with a value equal to l_p. The heapify phase is computed using Procedure $HEAPIFY()$, executed by all the internal — Actions *BackwardHeap*. The heapify phase eventually reaches the root which also executes Procedure $HEAPIFY()$ and initiates the third and last phase of our scheme, called the *Repair* phase — Action *InitRepair*. The aim of this phase is to correct the two following problems that

[4] To ease the reading of the algorithm, we assume that $S_p \in \{I, B\}$ (respectively, $\{I, H\}$) if p is the root (resp., p is a leaf). We could easily avoid this assumption by adding the following guarded action for the root (resp.leaf) node: $S_p = H$ (resp. $S_p = B$) $\longrightarrow S_p := I$. Note that this correction could occur only once.

Algorithm 2 Snap-Stabilizing PGCP Tree — Procedures.

```
1.01 Procedure HEAPIFY()
1.02     $C_p := C_p \cup \{\textbf{NEWNODE}\ (l_p, H, \{\})\}$
1.03     $l_p := \textbf{GCP}(\{l_c |\ c \in C_p\})$
1.04     for all $c \in C_p |\ l_c = l_p$ do
1.05         $C_p := C_p \cup C_c \setminus \{c\}$
1.06         $\textbf{DESTROY}(c)$
1.07     done

2.01 Procedure REPAIR()
2.02     while $\exists (c_1, c_2) \in C_p |\ l_{c_1} = l_{c_2}$ do
2.03         $C_p := C_p \cup \{\textbf{NEWNODE}(l_{c_1}, H, C_{s|\ s \in SameLabel(l_{c_1})})\}$
2.04         for all $c \in SameLabel_p(l_{c_1})$ do
2.05             $\textbf{DESTROY}(c)$
2.06         done
2.07     done
2.08     while $\exists c \in C_p |\ SamePGCP_p(l_c) \neq \emptyset$ do
2.09         $C_p := C_p \cup \{\textbf{NEWNODE}(l_c, H, C_c \cup SamePGCP_p(l_c))\}$
2.10         $C_p := C_p \setminus SamePGCP_p(l_c)$
2.11         $\textbf{DESTROY}(c)$
2.12     done
2.13     while $\exists (c_1, c_2) \in C_p |\ |GCP(l_{c_1}, l_{c_2})| > |l_p|$ do
2.14         $C_p := C_p \cup \{\textbf{NEWNODE}(GCP(l_{c_1}, l_{c_2}), H, SameGCP_p(GCP(l_{c_1}, l_{c_2}))\}$
2.15         $C_p := C_p \setminus SameGCP_p(GCP(l_{c_1}, l_{c_2}))$
2.16     done
```

can occur in the *PrefixHeap*. First, even if no node in T_p has the same label as p, the same label may exist in different branches of the tree; Second, if each node is the greatest common prefix of its children labels, it is not necessarily the greatest common prefix of any pairs of its children labels.

The *repair* phase works similarly as in the Broadcast phase. The root and the internal nodes execute Procedure $REPAIR()$ starting from the root toward the leaves — Actions *InitRepair* and $ForwardRepair$. During this phase, for each node p, four cases can happen:

1. Several children of p have the same label. Then, all the children with the same label are merged into a single child — Lines 2.02 to 2.07;
2. The labels of some children of p are prefixed with the label of some of its brothers. In that case, the addresses of the prefixed children are moved into the corresponding brother — Lines 2.08 to 2.12;
3. The labels of some children of p are prefixed with a label which does not exist among their brothers and which are longer than the label of p. Then, for each set of children with the same prefix, p builds a new node with the corresponding prefix label and the corresponding subset of nodes as children — Lines 2.13 to 2.16.
4. If none of the previous three cases appear, nothing is done.

Finally, Phase $REPAIR()$ ends at leaf nodes by executing Action *EndRepair*. This indicates the end of the PGCP tree construction. Note that since we are considering self-stabilizing systems, the internal nodes need to correct abnormal situations due to the unpredictable initial configuration. The unique abnormal situation which could avoid the normal progress of the three phases of our scheme is the following: An internal node p is in State B (done with its broadcast phase) but its father f_p is in State H or I, indicating that it is done executing its Heapify phase or it is Idle, respectively. In that case,

p executes Action *ErrorCorrection*, in the worst case, pushing down T_p the abnormal broadcast phase until reaching the leaf nodes of T_p. This guarantees the liveness of the protocol despite unpredictable initial configurations of the system.

4.2 Correctness proof

In this section we show that the algorithm described in Subsection 4.1 is a snap-stabilizing PGCP tree algorithm. The complexities are also discussed.

Remark 1. To prove that an algorithm provides a snap-stabilizing PGCP tree algorithm, we need to show that the algorithm satisfies the following two properties: (1) starting from any configuration, the root eventually executes an initialization action; (2) Any execution, starting from this action, builds a PGCP tree.

Let us first consider the algorithm by ignoring the two procedures $HEAPIFY()$ and $REPAIR()$. In that case, the algorithm is very similar to the snap-stabilizing PIF in [8]. The only difference between both algorithms consists in the third phase. In Algorithm 1, the third phase is initiated by the root only, *after* the heapify phase terminated only, whereas in [8], the third phase can be initiated by any node once itself and its father are done with the second phase. That means that with the solution in [8], both the second and the third phase can run concurrently. That would be the case with Algorithm 1 if the guard of Action *ForwardRepair* has been as follows: $S_p = H \wedge S_{f_p} \in \{H, I\} \wedge (\forall c \in C_p |\ S_c \in \{H, I\})$

However, it follows from the proofs in [8] that the behavior imposed by our solution is a particular behavior of the snap-stabilizing PIF algorithm. This behavior happens when all the nodes are slow to execute the action corresponding to the third phase. Since the algorithm in [8] works with an unfair daemon, the algorithm ensures that, eventually, the root initiates the third phase, leading the system to behave as Algorithm 1. Therefore, ignoring the effects of the two procedures $HEAPIFY()$ and $REPAIR()$ on the tree topology, the proof of snap-stabilization in [8] is also valid with our algorithm.

Considering the two procedures $HEAPIFY()$ and $REPAIR()$ again, since in every p, the set C_p is finite, it directly follows from the code of the two procedures in Algorithm 2 that that in every p, the set C_p is finite: every execution of Procedures $HEAPIFY()$ or $REPAIR()$ is finite.

It follows from the above discussion :

Lemma 1. *Starting from any configuration, the root node can execute Action InitBroadcost in a finite time even if the daemon is unfair.*

As a corollary of Lemma 1, the first condition of Remark 1 holds. Also, this show that every PGCP tree computation initiated by the root eventually terminates. It remains to show that the second condition of Remark 1 also holds for any node p.

Lemma 2. *After the execution of Procedure* **HEAPIFY** *by a node* p*,* T_p *is a PrefixHeap.*

Proof. We prove this by induction on $h(T_p)$. Since Procedure **HEAPIFY**() cannot be executed by a leaf node, we consider $h(T_p) \geq 1$.

1. Let $h(T_p)$ be equal to 1. So, all the children of p are leaves. Executing Lines 1.02 to 1.03, p is as a new child, itself a leaf node, labeled with l_p, while l_p contains the greatest common prefix of all its children. After the execution of Lines 1.04 to 1.07, p contains no child c such that $l_c = l_p$. Thus, l_p is a PGCP of all its children labels.
2. Assume that the lemma statement is true for any p such that $h(T_p) \leq k$ where $k \geq 1$. We will now show that the statement is also true for any p such that $h(T_p) = k + 1$. By assumption, the lemma statement is true for all the children of p, *i.e.,* $\forall c \in C_p$, l_c is a proper prefix of any label in T_c, and l_c is the PGCP of all nodes in C_c. So, after executing Procedure **HEAPIFY**(), following the same reasoning as in Case 1, l_p is a PGCP of all its children, and since themselves are the root of a PrefixHeap, for every $c \in C_p$, l_p is also a proper prefix of any label in T_c. Hence, the lemma statement is also true for p.

Corollary 1. *After the system executed a complete Heapify phase, the whole tree is a PrefixHeap.*

Lemma 3. *After the execution of Procedure* **REPAIR***() by a node* p *such that* $h(T_p) \geq 1$*, then for every pair* $(c_1, c_2) \in C_p$*,* $l_p = PGCP(c_1, c_2)$*.*

Proof. Given p such that $h(T_p) \geq 1$ and that l_p is a proper prefix of any l_c for $c \in C_p$ (what we know by Lemma 2), if the tree following conditions are true for every pair $(c_1, c_2) \in C_p$, the statement $\forall (c_1, c_2) \in C_p$, $l_p = PGCP(c_1, c_2)$ is true:

1. $l_{c_1} \neq l_{c_2}$;
2. l_{c_1} (resp. l_{c_2}) is not a prefix of l_{c_2} (resp. l_{c_1});
3. $|GCP(l_{c_1}, l_{c_2})| = |l_p|$.

Clearly, after the execution of Lines 2.02 to 2.07, Lines 2.08 to 2.12, and Lines 2.13 to 2.16, Conditions 1, 2, and 3 holds, respectively.

By induction of Lemma 3 on every node of the path from the root to each leaf node, we can claim:

Corollary 2. *After the system executed a complete Repair phase, the whole tree is a PGCP tree.*

Proof. By induction of Lemma 3 on every node of the path from the root to each leaf node.

From corollaries 1 and 2, and the fact that after the root executed Action $InitBroadcast$, the three phases $Broadcast$, $Heapify$, and $Repair$ proceed one after another [8], we can claim the following result:

Theorem 1. *Running under any daemon, Algorithm 1 and Algorithm 2 provide a snap-stabilizing Proper Greatest Common Prefix Tree construction.*

4.3 Complexity

Theorem 2. *The average time for the PGCP tree construction is $O(h + h')$ rounds. In the worst case, the construction requires $O(n)$ space complexity, $O(n)$ rounds and $O(n^2)$ operations, where n is the number of nodes of the tree.*

Proof. By similarity with the PIF, we can easily establish that the *broadcast* phase has reached all leaf nodes in $O(h)$ rounds, where h is the height of the tree when the *InitBroadcast* action is performed. We also easily see that the *heapify* phase reaches the root in $O(h)$. During the *repair* phase, the number of rounds required to reach all leaf nodes of the repaired tree (and thus end the cycle) is clearly $O(h')$, where h' is the height of this repaired tree (each round increment the depth by 0 or 1). The first part of the theorem is established.

When the *repair phase* is initiated, more precisely after the execution of the **HEAPIFY** macro and before the execution of the **REPAIR** macro on the root, it may happen that the tree becomes a star graph, each node being a child of the root (obviously except the root itself). This case is clearly the worst case, not only in terms of extra space required ($n - 1 = O(n)$) but also in terms of number of operations since the complexity of the **REPAIR** macro depends on the number of the root's children, *i.e.,* also $n - 1$. More precisely, the **REPAIR** macro is a combination of three operations: merging nodes, lines 2.02 to 2.06, moving nodes under other ones, lines 2.08 to 2.12 or creating a new subtree, lines 2.13 to 2.16. Among the set of possible combinations, the one that leads to the weakest parallelism is the move of $n - 2$ children of the root under a given node s, since, in the next round, s will be the only one process to work, *i.e.,* process these $n - 2$ nodes. If this worst case repeats (and the final topology is a chain), the complexity is of the following shape:

$$a \times (n - 1) + a \times (n - 2) + \ldots + a = O(n^2)$$

where a is a constant. Even if the worst case is not really attractive, we use simulations in the next section to see what we can expect in real life in terms of latency and extra space.

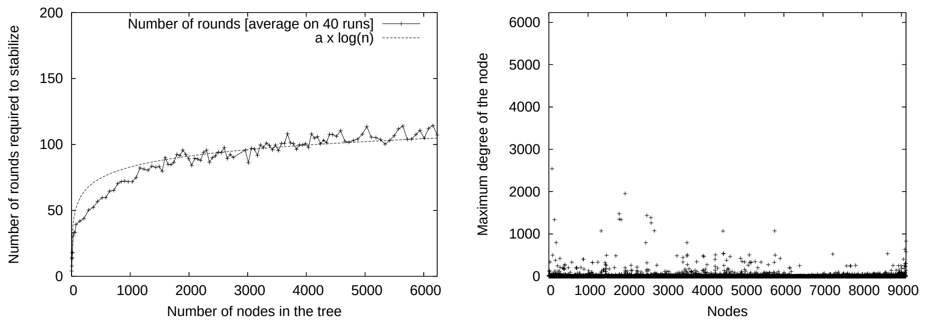
5 Simulation results

To better capture the expected behavior of the snap-stabilizing PGCP tree, we simulated the algorithm using relevant data sets which reflect the use of computational platforms. The simulator is written in Python and contains the three following main parts:

1. It creates the tree with a set of labels of basic computational services commonly used in computation grids such as the names of routines of linear algebra libraries, the names of operating systems, the processors used in today's clusters and the nodes' addresses. The number of keys is up to 5200, creating trees up to 6228 nodes (the final tree size is the number of labels

inserted plus the number of labels created to reflect the prefix patterns). For instance, inserting two labels `DTRSM` and `DTRMM` results in a tree whose root (common father of `DTRSM` and `DTRMM`) is labeled by `DTR`.

2. It destroys the tree by moving subtrees, randomly. This is achieved by modifying the father of a randomly picked node and moving it from the set of children of its father to the set of children of a randomly chosen node. This operation is repeated up to $n/2$ nodes (meaning that, in average, approximately $n/2$ nodes are connected to a wrong father).
3. It launches the algorithms by testing for each node if the state of the node and those of its neighbors satisfy the guard of some action in the algorithm, in which case the statement of the action is executed. This step is repeated until the tree is in a stable configuration, *i.e.,* a configuration where all nodes are in state I again.



(a) Number of Rounds. (b) Highest Degrees.

Fig. 1. Simulation of the snap-stabilizing PGCP tree.

We have first collected results on the latency of the algorithm. Figure 1-(a) gives the average number of rounds required to have a stable configuration, starting from 40 different bad configurations. The tree size ranges between 2 and 6228. We observe that the number of rounds required by the algorithm has a logarithmic behavior (and not linear as previously suggested by the worst case). It clearly scales according to the height of the tree, thus confirming the average complexity of the algorithm and its good scalability.

We have also collected results on the extra space required on each node. Since the tree topology undergoes changes during the reconstruction, degrees of nodes also dynamically change as nodes are created, destroyed, merged or moved. Figure 1-(b) shows the highest degree of nodes, *i.e.,* the real extra space required on each node, including nodes created and/or destroyed during the reconstruction. The final tree size is 6228; the total number of nodes, including temporary nodes, is 9120. The experiment shows that the highest of maximum degree of all nodes is 2540, and most of maximum degrees are very low (less than 50). This can be partly explained by the fact that the deepest a node is, the smaller is its degree. In other terms, during a breadth-first traversal of the tree, the topology quickly enlarges close to the root and then its breadth remains relatively stable until reaching the leaf nodes. More generally, this simulation shows that the worst case is far to be reached and that only few nodes will require an large extra space.

6 Conclusion

This paper presents the first snap-stabilizing greatest common prefix tree and a general self-stabilization algorithm for distributed tries. It provides an alternative to tree-structured peer-to-peer networks suffering from the high cost of replication mechanisms and a first step of an innovating way to reach the fault tolerance requirements over large distributed systems. Our algorithm is optimal in terms of stabilization time since we prove it to be snap-stabilizing. It requires in average a number of rounds proportional to the height of the tree, thus providing a good scalability. This result has been confirmed by simulation experiments based on relevant data sets. On the theoretical side, our future work will consist to improve the worst case complexities in terms of extra space requirements and total latency. Also, note that our model assumes that the processes can communicate with each other. In the state model, this is modeled as if every process can read the variables of all the processes of the network. However, once implemented in the message-passing model, the protocol requires communications between processes involved in the tree only. So, on the experimental side of our future works, we plan to implement this algorithm in the message-passing with a model based on that introduced in [18]. On this other hand, we also plan to implement our algorithm inside a prototype of a peer-to-peer indexing system we are currently developing, based on the JXTA toolbox. First experiments have been conducted on the Grid'5000 platform [9].

References

1. A. Andrzejak and Z. Xu. Scalable, Efficient Range Queries for Grid Information Services. In *Peer-to-Peer Computing*, pages 33–40, 2002.
2. A. Arora and M.G. Gouda. Distributed Reset. *IEEE Transactions on Computers*, 43:1026–1038, 1994.
3. J. Aspnes and G. Shah. Skip Graphs. In *Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–393, January 2003.
4. B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. Time Optimal Self-stabilizing Synchronization. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC '93)*, pages 652–661, 1993.
5. M. Balazinska, H. Balakrishnan, and D. Karger. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In *International Conference on Pervasive Computing 2002*, 2002.
6. S. Basu, S. Banerjee, P. Sharma, and S. Lee. NodeWiz: Peer-to-Peer Resource Discovery for Grids. In *5th International Workshop on Global and Peer-to-Peer Computing (GP2PC)*, May 2005.

7. D. Bein, Datta A.K., and Villain V. Snap-Stabilizing Optimal Binary Search Tree. In Springer LNCS 3764, editor, *Proceedings of the 7th International Symposium on Self-Stabilizing Systems (SSS '05)*, pages 1–17, 2005.
8. A. Bui, A. Datta, F. Petit, and V. Villain. State-optimal snap-stabilizing pif in tree networks. In IEEE, editor, *Proceedings of the 4th International Workshop on Self-Stabilizing Systems*, pages 78–85, 1999.
9. F. Cappello *et al.* Grid'5000: a Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform. In *SC'05: Proc. The 6th IEEE/ACM International Workshop on Grid Computing Grid'2005*, pages 99–106, Seattle, USA, November 13-14 2005. IEEE/ACM.
10. E. Caron, F. Desprez, C. Fourdrignier, F. Petit, and C. Tedeschi. A Repair Mechanism for Tree-structured Peer-to-peer Systems. In Springer Verlag, editor, *The Twelfth International Conference on High Performance Computing, HiPC 2006*, LNCS, Bangalore, India., December 18-21 2006.
11. E. Caron, F. Desprez, and C. Tedeschi. A Dynamic Prefix Tree for the Service Discovery Within Large Scale Grids. In A. Montresor, A. Wierzbicki, and N. Shahmehri, editors, *The Sixth IEEE International Conference on Peer-to-Peer Computing, P2P2006*, pages 106–113, Cambridge, UK., September 6-8 2006. IEEE.
12. N.S. Chen, H.P. Yu, and S.T. Huang. A Self-stabilizing Algorithm for Constructing Spanning Trees. *Information Processing Letters*, 39:147–151, 1991.
13. A. Cournier, A. K. Datta, F. Petit, and V. Villain. Enabling Snap-Stabilization. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 12, Washington, DC, USA, 2003. IEEE Computer Society.
14. A. Datta, M. Hauswirth, R. John, R. Schmidt, and K. Aberer. Range Queries in Trie-Structured Overlays. In *The Fifth IEEE International Conference on Peer-to-Peer Computing*, 2005.
15. E. W. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Commun. ACM*, 17(11):643–644, 1974.
16. S. Dolev. *Self-Stabilization.* The MIT Press, 2000.
17. S. Dolev, A. Israeli, and S. Moran. Self-stabilization of Dynamic Systems Assuming only Read/Write Atomicity. *Distributed Computing*, 7:3–16, 1993.
18. T. Herault, P. Lemarinier, O. Peres, L. Pilard, and J. Beauquier. A model for large scale self-stabilization. In IEEE Sc., editor, *21th International Parallel and Distributed Processing Symposium (IPDPS 2007)*, 2007.
19. T. Herman and Pirwani I. A Composite Stabilizing Data Structure. In Springer LNCS 2194, editor, *Proceedings of the 5th International Workshop on Self-Stabilizing Systems*, pages 197–182, 2001.
20. T. Herman and Masuzawa T. A Stabilizing Search Tree with Availability Properties. In IEEE, editor, *Proceedings of the 5th International Symposium on Autonomous Decentralized Systems (ISADS'01)*, pages 398–405, 2001.
21. T. Herman and Masuzawa T. Available Stabilzing Heaps. *Information Processing Letters*, 77:115–121, 2001.
22. A. Iamnitchi and I. Foster. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In *IPTPS*, pages 118–128, 2003.
23. D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Distributed Resource Discovery on PlanetLab with SWORD. In *Proceedings of the ACM/USENIX Workshop on Real, Large Distributed Systems (WORLDS)*, December 2004.
24. S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Prefix Hash Tree An indexing Data Structure over Distributed Hash Tables. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, St. John's, Newfoundland, Canada, July 2004.

25. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Adressable Network. In *ACM SIGCOMM*, 2001.
26. A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-To-Peer Systems. In *International Conference on Distributed Systems Platforms (Middleware)*, November 2001.
27. C. Schmidt and M. Parashar. Enabling Flexible Queries with Guarantees in P2P Systems. *IEEE Internet Computing*, 8(3):19–26, 2004.
28. Y. Shu, B. C. Ooi, K. Tan, and Aoying Zhou. Supporting Multi-Dimensional Range Queries in Peer-to-Peer Systems. In *Peer-to-Peer Computing*, pages 173–180, 2005.
29. I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup service for Internet Applications. In *ACM SIGCOMM*, pages 149–160, 2001.
30. P. Triantafillou and T. Pitoura. Towards a Unifying Framework for Complex Query Processing over Structured Peer-to-Peer Data Networks. In *DBISP2P 2003*, September 2003.