Self-Stabilization in Tree-Structured Peer-to-Peer Service Discovery Systems*

Eddy Caron†, Ajoy K. Datta°, Franck Petit‡ and Cédric Tedeschi†
†University of Lyon. LIP Laboratory. UMR CNRS - ENS Lyon - INRIA - UCB Lyon 5668. France
° School of Computer Science. University of Nevada Las Vegas. USA
‡ MIS Laboratory. University of Picardie Jules Verne. France

Abstract

The efficiency of service discovery is critical in the development of fully decentralized middleware intended to manage large scale computational grids. This demand influenced the design of many peer-to-peer based approaches. The ability to cope with the expressiveness of the service discovery was behind the design of a new kind of overlay structures that is based on tries, *or* prefix trees. *Although these overlays are well designed, one of their weaknesses is the lack of any concrete fault tolerant mechanism, especially in dynamic platforms; the faults are handled by using preventive and costly mechanisms,* e.g., *using a high degree of replication. Moreover, those systems cannot handle any arbitrary transient failure.*

Self-stabilization, *which is an efficient approach to design reliable solutions for dynamic systems, was recently suggested to be a good alternative to inject fault-tolerance in peer-to-peer systems. However, most of the previous research on self-stabilization in tree and/or P2P networks was designed in theoretical models, making these approaches hard to implement in practice. In this paper, we provide a self-stabilizing message passing protocol to maintain prefix trees over practical peer-to-peer networks. A complete correctness proof is provided, as well as simulation results to estimate the practical impact of our protocol.*

Keywords*: fault-tolerance, peer-to-peer systems, service discovery, self-stabilization.*

1 Introduction

Grids connecting geographically distributed computing resources have become a low cost alternative to supercomputers. The communities of grid computing and peer-to-peer together developed new approaches to design grid middleware over fully decentralized platforms [19], especially dealing with resource discovery. The demand for flexibility and ability to handle complexity of the service discovery systems led to the development of various overlay structures. One of them is based on *tries a.k.a.* lexicographic trees or prefix trees. These architectures usually support range queries, automatic completion of partial search strings, and are easy to extend to multi-attribute queries.

Although fault-tolerance is a mandatory feature of systems targeted for large scale platforms (to avoid data loss and to ensure proper routing), tries overlays offer only a poor robustness in dynamic environment. The crash of one or more nodes may lead to the loss of stored objects, and may split the trie into several subtries. These subtries may not be re-grouped correctly, making the system unable to correctly process queries. In recent trie-based approaches, the fault-tolerance has been either ignored, or handled by preventive mechanisms, usually by replication, which can be very costly in terms of computing and storage resources. Moreover, replication does not ensure the recovery of the system from arbitrary failures.

The concept of self-stabilization [11, 12] is a general technique to design distributed systems that can handle arbitrary transient faults. A self-stabilizing system, regardless of the initial states of the process and initial messages in the links, is guaranteed to converge to the intended behavior in finite time.

Related Work. The resource discovery in P2P environments has been extensively studied. Although DHTs [22, 23, 27, 28] were designed for very large systems, they provide only rigid mechanisms of search. A great deal of research went into finding ways to improve the retrieval process over structured peer-to-peer networks. Peer-to-peer systems use different technolo-

*Partially funded by ANR (Agence Nationale de la Recherche) through the LEGO project ANR-05-CIGC-11.

gies to support multi-attribute range queries [3, 20, 24, 26]. 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 several branches of the trie.

Among trie-based approaches, Prefix Hash Tree (PHT) [21] dynamically builds a trie of the given keyspace as an upper layer, and maps it over any DHT-like network. The architecture of PHT increases the complexity of the trie and of the underlying DHT. The problem of fault-tolerance is then delegated to the DHT layer. Skip Graphs, introduced in [1], are similar to tries, but rely on skip lists, using probabilistic fault-tolerance. The fault-tolerance approach used in P-Grid [9] is based on probabilistic replication.

In this paper, we focus on the DLPT (Distributed Lexicographic Placement Table), an architecture recently developed and studied in [8]. This approach, initially designed for the purpose of service discovery over dynamic computational grids and aimed at solving some drawbacks of these previous approaches, is a two layer architecture. The upper layer is a prefix tree maintaining the information about available services; each node of this tree maintains the information about services sharing a particular key. This key labels the node. This tree is mapped onto the lower layer, *i.e.*, the physical networks, for example, using a distributed hash table. In its early design, the fault-tolerance was addressed by replication of nodes and links of the tree. An advantage of this technology is its ability to take into account the heterogeneity of the underlying physical network to builid a more efficient tree overlay [8].

In summary, the fault-tolerance in resource discovery systems was ignored, delegated to other layers, or implemented using replication. In [6], a first alternative to the replication approach is provided; this scheme handles the trie crash by reconnecting and reordering the nodes. However, the subtries being reordered are assumed to be valid, restricting the initial configurations being handled and repaired, and making this solution not self-stabilizing. In the self-stabilizing area, some investigations take interest in distributed search tree overlay, *e.g.*, [17] and [18] for 2-3 trees and heap trees, respectively. A self-stabilizing lexicographic distributed structure is proposed in [15] as an application of r-operators. Some papers considered self-stabilization in a peer-to-peer setting [7, 10, 13, 16, 25]. In [16], a self-stabilizing spanning tree protocol is proposed. In the same paper, the authors introduce a new model of peer-to-peer systems for self-stabilization. In [13], a self-stabilizing protocol that maintains a hypertree is proposed. The protocol proposed in [7] deals with prefix tree maintenance. It has the nice property of being snap-stabilizing [4], *i.e.*, it guarantees that it always behaves according to its specification — it is a self-stabilizing algorithm which is optimal in terms of stabilization time since its stabilizes in 0 steps. However, the algorithms in [7] requires the initial topology to be a rooted connected tree, and has been proven in a coarse theoretical model (the state model introduced in [11]).

Contributions. In this paper, we propose a self-stabilizing protocol to maintain a prefix tree in a message passing *peer-to-peer oriented* model. Our algorithm does not require a fixed root node and works with any arbitrary initial configuration of the tree topology. The proposed protocol can be implemented on any platform that supports message passing and basic services available in most peer-to-peer systems. We give a formal proof of correctness of our protocol and some simulation results to study the scalability of the protocol as well as its efficiency in keeping the architecture available for clients even under high failure rate. In Section 2, we give the model in which our protocol is designed, and the data structures it maintains. The protocol is given in Section 3, followed by its proof in Section 4. Simulation results are given in Section 5.

2 Preliminaries

The network. A P2P network consists of a set of asynchronous processors with distinct ids. The processors communicate by exchanging messages. Any processor P_1 can communicate with another processor P_2 provided P_1 knows the id of P_2. We abstract the details of the actual routing, as it is done in most of the peer-to-peer systems. Henceforth, we use the word *peer* to refer to a processor.

The indexing logical tree. Our indexing system is a logical prefix tree, whose nodes are mapped onto the peers of the network. Henceforth, the word *node* refers to a node of the tree, *i.e.*, a logical entity. Keep in mind that *Physical nodes*, or *processors* are referred to as *peers*. Each peer maintains a part of our indexing system, *i.e.*, some (*logical*) nodes of the prefix tree. It is also important to recall that a (*logical*) node is implemented as a process, which runs on a peer. In other words, a process implements the notion of node. As we said, each node has a label. In a correct configuration (that we define later in Definition 1), each node label is

unique. (Only one node is responsible for all services sharing a common key.) However, initially, when the system is not yet stabilized, the structure may contain multiple nodes sharing the same label. Thus, we cannot use labels to identify the nodes. We chose to identify nodes by the process implementing it. A process is identified by a unique combination of the peer running it and a port number. Our protocol maintaining the prefix tree is run on every process. *Nodes* and *Processes* are basically two different view of the same thing. In the remainder, we use these terms interchangeably. Note that the tree topology is susceptible to changes during its reconstruction. We assume the presence of a service able to return process references. This *process discovery* service is similar to the one used in [16]. Any process of the system can obtain any other process identifier by calling this service. To prove the correctness of the algorithm, we assume that a finite number of queries to this service is enough to collect the identifiers of all processes in the system. The service provides the following two primitives: GETRUNNINGPROCESS() returns the identifier of a randomly chosen process, and GETNEWPROCESS() creates a new process (without setting its parameters yet) and returns its identifier. The communication between processes is carried out by exchanging messages. A process p is able to communicate with a process q, if and only if p knows the id of q. We assume that a copy of every message sent by p to q is eventually received by q, unless q has crashed or has been killed. The message delay is finite but not bounded. Messages arrive in the order they were sent (FIFO), and as long as a message is not processed by the receiving process, we assume that it is in transit.

Proper Greatest Common Prefix Tree. We now formally describe the distributed structure we maintain. 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 as $u \circ v$, 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 1 (PGCP Tree) *A* Proper Greatest Common Prefix Tree *is a labeled rooted tree such that the following properties are true for every node of the tree:*

1. *The node label is a proper prefix of any label in its subtree.*
2. *The greatest common prefix of any pair of labels of children of a given node are the same and equal to the node label.*

Self-stabilization. Define a *transition system* as a triple $\mathcal{S} = (\mathcal{C}, \mapsto, \mathcal{I})$, where $\mathcal{C}$ is a set of configurations, $\mapsto$ is a binary transition relation on $\mathcal{C}$, and $\mathcal{I} \subset \mathcal{C}$ is the set of initial configurations. A *configuration* is a vector with $n + 1$ components, where the first n components are the state of n processes and the last one is a multi-set of messages in transit in m links. We define an *execution* of $\mathcal{S}$ as a maximal sequence $\mathcal{E} = (\gamma_0, \gamma_1, \gamma_2, ..., \gamma_i, \gamma_{i+1}, ...)$, where $\gamma_0 \in \mathcal{I}$ and for $i \geq 0, \gamma_i \mapsto \gamma_{i+1}$. A predicate Π on $\mathcal{C}$, the set of system configurations, is *closed* for a transition system S if and only if every state of an execution e that starts in a state satisfying Π also satisfies Π. A transition system S is *self-stabilizing* with respect to a predicate Π if and only if Π is closed for S and for every execution e of S, there exists a configuration of e for which Π is true.

3 Protocol

In this section, we present a protocol to build a PGCP tree providing a logical overlay structure of a peer-to-peer service discovery system. Our protocol is self-stabilizing and self-organzing, meaning, the logical structure can start from an arbitrary state, but will eventually converge to a PGCP tree. The proposed protocol assumes the existence of an underlying *self-stabilizing end-to-end communication* (SSEE) protocol. Both layers communicate using **send**/**receive** primitives over FIFO message queues. The "**send**(<m>, q)" primitives sends message m to node q; it always terminates; if the recipient q is alive, m is queued at q and will be processed later; otherwise, q crashed and m is lost. The implementation of Protocol SSEE is beyond the scope of this paper. Refer to [2, 14] for such protocols.

Every process p (we use p to denote the id of p and the address used by other processes to communicate with p) has a label l_p. Denote by $\widehat{p}$, the pair (p, l_p).

Recall that, $\widehat{p} = \widehat{p'}$ is equivalent to $(p = p') \wedge (l_p = l_{p'})$. Node p also maintains a copy of the identifier and label of its parent into $\widehat{f_p}$ and of its children into the finite set $\widehat{C_p}$. Note that $\widehat{C_p}$, $\hat{p}$ and $\widehat{f_p}$ are variables, C_p is the result of a macro that extracts the first element of every pairs in $\widehat{C_p}$.

To deal with crash failures, to maintain the topological information, and to maintain the status of processes (they terminated or not) in our protocol, we assume the presence of an underlying *heartbeat* protocol. We assume that any node that does not receive news from one child or parent during a bounded time (implemented by using a TimeOut action in the algorithm), removes the node from its neighborhood set. Note that when deleting a given child $q \in C_p$, all the data associated with q is deleted.

The function GETEPSILON() returns the identifier of a random node labeled by the empty word ϵ. It relies on the *process discovery* service previously described. Basically, it calls GETNEWPROCESS() and checks if the process returned is an ϵ-process, *i.e.*, a process labeled by ϵ. Since we assume that a finite number of calls to the *process discovery* service is enough to get all identifiers of alive nodes, the `GetEpsilon()` function also returns every ϵ-process in a finite time. The NEWPROCESS(lbl, f, C) function starts a new process on the local node and initiates the label with lbl, the parent with f, and the set of children with C.

The "**send**($<m>$, q)" procedure returns a Boolean. If it returned `true`, the recipient q is alive and the message m is queued on q, and will be processed later. Otherwise, it means that q is dead. The rules of the protocol, the *periodic* rule, periodically runs on each node as detailed in Algorithm 1, and the *upon receipt* rules, detailed in Algorithm 2, are initiated upon receipt of a message, are atomic.

Each node p periodically initiates the action described by Algorithm 1. p begins by eliminating the cases where p is either a parent or a child of itself (this may cause cycles) (Lines 2.02-2.03).

Lines 2.04-2.13 deal with parent maintenance. These lines ensure that eventually, there will be one and only one root, *i.e.*, only one node p eventually satisfies $f_p = \perp$. To achieve this, the possible root nodes merge. Let us consider a root node p to explain this part of the algorithm. There are two possible situations:

1. If the label of p is ϵ, p tries to connect to another node q, also labeled ϵ. q then becomes a child of p (Line 2.08). p informs q that its parent changed using UPDATEPARENT message. Upon receipt of that message, q updates its parent variable (Lines 6.01-6.03 of Algorithm 2). Since p and q are labeled identially, they will merge (the merge process is explained below), thus reducing the number of roots by one.

2. If p is not labeled by ϵ, a new node labeled ϵ is artificially created as the parent of p. This new node executes the periodic rule satisfying the previous case.

Lines 2.15-2.27 deal with children maintenance to make sure that eventually, every set of children satisfies Definition 1. This phase consists of three parts.

1. We eliminate cases where the set of children of p contains a node q whose label is the label of p by initiating the merge process of p and q. p sends a MERGE message to q (Lines 2.15-2.16). First, upon receipt of the MERGE message, q informs its children that their new parent is their current grandparent through GRANDPARENT messages. Upon receipt of this message, the children of q change their parent from q to p. To ensure a good synchronization, q waits until all its children have been accepted by p as children, *i.e.*, waits for the GFDONE message. q finally informs p that the merging process has finished by sending the MDONE message, and terminates (Lines 8.01-11.03).

Algorithm 1 Periodic rule, on process p

```
1.01 Variables:  p̂ = (p, l_p), id and label of p
                 f̂_p = (f_p, l_{f_p}), id and label of the parent of p
                 Ĉ_p = {q̂_1 = (q_1, l_{q_1}), ..., q̂_k = (q_k, l_{q_k})}, set of children of p
                 T_q, ∀q ∈ C_p time before considering q as not its child anymore
                 C_p ≡ {q | (q, l_q) ∈ Ĉ_p}, set of totally ordered ids of children of p

2.01 Upon TimeOut do
2.02     if f_p = p then f̂_p := ⊥
2.03     if p ∈ C_p then Ĉ_p := Ĉ_p \ {p̂}
2.04     if f_p = ⊥ then
2.05         if l_p = ε then
2.06             q := GETEPSILON()
2.07             if q ≺ p then
2.08                 Ĉ_p := Ĉ_p ∪ {(q, ε)}
2.09                 send(<UPDATEPARENT, p̂>, q)
2.10         else
2.11             new := GETNEWPROCESS()
2.12             send(<HOST, (ε, ⊥, {p̂})>, new)
2.13             f̂_p := (new, ε)
2.14
2.15     while ∃q ∈ C_p | l_q = l_p do
2.16         send(<MERGE, p̂>, q)
2.17     while ∃(q_1, q_2) ∈ C_p^2 : l_p ∈ PREFIXES(l_{q_1}) ∧ l_{q_1} ∈ PREFIXES(l_{q_2})) do
2.18         send(<UPDATEPARENT, q̂_1>, q_2)
2.19         Ĉ_p := Ĉ_p \ {q̂_2}
2.20     while ∃(q_1, q_2) ∈ C_p^2 : l_p ∈ PREFIXES(l_{q_1})
2.21         ∧l_p ∈ PREFIXES(l_{q_2}) ∧ |GCP(l_{q_1}, l_{q_2})| > |l_p| do
2.22         l_new := GCP(l_{q_1}, l_{q_2})
2.23         new := GETNEWPROCESS()
2.24         send(<HOST, (l_new, p, {q_1, q_2})>, new)
2.25         send(<UPDATEPARENT, n̂ew>, q_1)
2.26         send(<UPDATEPARENT, n̂ew>, q_2)
2.27         Ĉ_p := Ĉ_p \ {q̂_1, q̂_2} ∪ {n̂ew}
2.28     if f_p ≠ ⊥ then
2.29         send(<PARENT?, p̂>, f_p)
```

Algorithm 2 *Upon receipt* rules, on process p

```
3.01  upon receipt of <PARENT?, q̂> do
3.02      if l_p ∈ PREFIXES(l_q) ∧ send(<CHILD, p̂>, q) then
3.03          Ĉ_p := Ĉ_p ∪ {q̂}
3.04      else
3.05          send(<ORPHAN, p̂>, q)
3.06          Ĉ_p := Ĉ_p \ {q̂}

4.01  upon receipt of <CHILD, q̂> do
4.02      if f_p = q then
4.03          l_{f_p} := l_q

5.01  upon receipt of <ORPHAN, q̂> do
5.02      if f_p = q then
5.03          f_p := ⊥

6.01  upon receipt of <UPDATEPARENT, q̂> do
6.02      if (l_q ∈ PREFIXES(l_p)) ∧ send(<PARENT?, p̂>, q) then
6.03          f̂_p := q̂

7.01  upon receipt of <HOST, l, f, Ĉ> do
7.02      NEWPROCESS (l, f, Ĉ)

8.01  upon receipt of <MERGE, q̂> do
8.02      if (f_p = q) ∧ (l_q ∈ PREFIXES(l_p)) then
8.03          ∀q' ∈ C_p, send(<GRANDPARENT, f̂_p, q̂>, q')

9.01  upon receipt of <GRANDPARENT, n̂ewf, q̂> do
9.02      if (f_p = q) ∧ (l_q ∈ PREFIXES(l_p)) then
9.03          f̂_p := n̂ewf
9.04          send(<GFDONE, p̂>, q)

10.01 upon receipt of <GFDONE, q̂> do
10.02     if (q ∈ C_p) ∧ (l_p ∈ PREFIXES(l_q)) then
10.03         Ĉ_p := Ĉ_p \ {q̂}
10.04         if Ĉ_p = ∅ then
10.05             send(<MDONE, p̂>, f_p)
10.06             KILL(p)

11.01 upon receipt of <MDONE, q̂> do
11.02     if (q ∈ C_p) ∧ (l_p ∈ PREFIXES(l_q)) then
11.03         Ĉ_p := Ĉ_p \ {q̂}
```

2. We eliminate cases where a pair of children do not satisfy Definition 1. For example, assume that a child q_1 prefixes another child q_2. So, the proper greatest common prefix of the labels of q_1 and q_2 is equal to the label of q_1. But, the greatest common prefix, by Definition 1 must be the label of p. A contradiction (Line 2.17). q_2 then becomes the child of q_1 (Lines 2.17-2.19).

3. We check that there is no pair (q_1, q_2) in its set of children such that the greatest common prefix g of their labels is greater than its own label (Lines 2.21-2.27). In this case, a new node must be created. This node, labeled by g, will be the child of p and the common parent of q_1 and q_2.

The purpose of Lines 2.28-2.29 is for p to check the validity of its parent. Upon receipt of the PARENT message, the parent of p decides whether p is its child depending on their labels, and informs p of the result. It uses a CHILD message to indicate that it considers p as its child. Otherwise, it sends an ORPHAN message. Lines 4.01-5.03 detail the receipt of these messages. Upon receipt of CHILD, p updates the label of its parent. Upon receipt of ORPHAN, it becomes a root and executes the periodic rule as we discussed before.

4 Proof of Stabilization

Like most of the fault-tolerant system design schemes, we will assume the following; (i) The frequency of fault occurrence is not too high. (ii) The time between two occurrences of faults is higher than the time required to recover from a fault.

In the proofs given in this section, we will consider a suffix of an execution starting after all crashes have taken place, *i.e.*, in this particular execution segment, no further crashes will occur. Let P be the set of alive processes. Every "**send**($<m>$, q)" executed by a process $p \in P$ terminates and when this happens, either m is received by q or $q \notin P$.

A configuration γ satisfies Predicate Π_1 if and only if, assuming that a process $p \in P$ infinitely often sends a message to a process $q \in P$ ($q \neq p$), the following two conditions are true in every execution e starting from γ: (1) *Safety*: The sequence of messages received by q is a prefix of the sequence of messages sent by p. (2) *Liveness*: q receives a message infinitely often.

The following lemma follows from the fact that we assume an underlying self-stabilizing end-to-end communication protocol.

Lemma 2 *The system is self-stabilizing with respect to* Π_1.

Corollary 3 *Every message received by a process in* P *in a configuration that satisfies* Π_1, *was sent by another process in* P.

Lemma 4 *Starting from a configuration satisfying* Π_1, *every process in* P *executes Lines 2.01-2.29 infinitely often.*

Proof. The set $\widehat{C_p}$ is finite and the loop is executed atomically (no message receipt can interrupt the execution of the loop). Moreover, each execution of **send** terminates. So, none of the three "while loops" (Lines 2.15-2.27) can loop forever. □

Corollary 5 *Starting from a configuration satisfying* Π_1, *the system eventually contains no process* p *such that* $f_p = p$ *or* $p \in C_p$.

Proof. Process p executes Lines 2.02 and 2.03 infinitely often. Moreover, the algorithm contains no line in which $f_p := p$ or $\widehat{C_p} := \widehat{C_p} \cup \{\widehat{p}\}$. □

We will now show that, starting from a configuration satisfying Π_1, the child set (C_p) of each process $p \in P$ eventually contains no child Id q such that $q \notin P$, *i.e.*, q is alive.

Lemma 6 *Let p be a process in P. In every execution starting from a configuration γ satisfying Π_1, if there exists $q \in C_p$ such that $q \notin P$, then eventually, $q \notin C_p$.*

Proof. Follows from the assumption of an underlying *heartbeat* protocol between neighbors. □

Let Π_2 be the predicate over $\mathcal{C}$ such that $\gamma \in \mathcal{C}$ satisfies Π_2 iff $\forall p \in P, \forall q \in C_p, q \in P$.

Lemma 7 *The system is self-stabilizing with respect to* Π_2.

Proof. From Corollary 3, no process $p \in P$ can receive a message from a process $q \notin P$. So, in any execution starting from a configuration γ satisfying Π_1, no process p can add a process Id q such that $q \notin P$. p can add a process q in C_p using Lines 2.08 and 2.27 in which case q was returned by a GET*() function assumed to return ids in P. It can also add q using Line 3.03, in which case q was sent by q itself, and is thus alive. By Lemma 6, if there exists some process $p \in P$ such that C_p contains ids not in P, then each of these ids is eventually removed from C_p. Thus, eventually, $\forall p \in P$, $\forall q \in C_p, q \in P$. □

From now on, we do not mention P because all process references are assumed to be in P. Let Π_3 be the predicate over $\mathcal{C}$ such that $\gamma \in \mathcal{C}$ satisfies Π_3 iff in every execution starting from γ satisfying Π_2, for each process p: (1) p executed Lines 2.01-2.29 at least once, and (2) if p sent a message "PARENT?" to q, then p received the corresponding response (a message <CHILD> or <ORPHAN>) from q. The next lemma follows directly from Lemma 4 and the fact that every message receipt action terminates.

Lemma 8 *The system is self-stabilizing with respect to* Π_3.

Lemma 8 ensures that for every process p, each label in $\widehat{C_p}$ is correct, *i.e.*, is equal to the actual label of q. p adds or updates the child labels in three ways. First, using Line 3.03 in which case the label was sent by q itself and is thus correct. Second, by using Line 2.08 in which case q was returned by GETEPSILON() and the label is set to ϵ. Third, by using Line 2.27 in which case the label was computed by p itself and then sent to new. We will now show that, starting from a configuration satisfying Π_2, the child set (C_p) of each process p eventually contains no child Id q such that $l_p \notin \text{PREFIXES}(l_q)$.

Lemma 9 *Let p and q be two processes. If there exists an execution starting from a configuration satisfying Π_3 containing a system transition $\gamma_t \mapsto \gamma_{t+1}$ such that $q \notin C_p$ in γ_t and $q \in C_p$ in γ_{t+1}, then $l_p \in \text{PREFIXES}(l_q)$.*

Proof. To add q to C_p, p executes one of the following lines:

1. Line 2.08. In this case, $l_p = l_q = \epsilon$.
2. Line 2.27. In this case, $q = new$ and $l_q = GCP(l_{q_1}, l_{q_2})$, where both l_{q_1} and l_{q_2} are prefixed by l_p.
3. Line 3.03. This line is executed only if $l_p \in \text{PREFIXES}(q)$ (Line 3.02).

□

Lemma 10 *Let γ be a configuration satisfying Π_3. Let p and q be a pair of processes such that, in γ, $q \in C_p$. If there exists an execution e starting from γ such that $q \in C_p$ forever, then $l_p \in \text{PREFIXES}(l_q)$.*

Proof. Assume by contradiction that there exists e starting from γ such that $q \in C_p$ forever, and $l_p \notin \text{PREFIXES}(l_q)$. There are two cases to consider:

1. There exists a configuration $\gamma' \in e$ such that $f_q \neq p$ forever ($f_q \neq p$ in every execution starting from γ'). In that case, assuming the presence of an underlying *heartbeat* protocol between neighbors, p will not receive heartbeats from q and eventually remove it from the set of its children. A contradiction.
2. $f_q = p$ infinitely often. So, q sends PARENT? to p infinitely often. Upon receipt of this message, p removes q from C_p (Line 3.06). A contradiction.

□

Let Π_4 be the predicate over $\mathcal{C}$ such that $\gamma \in \mathcal{C}$ satisfies Π_4 iff given two processes p, q, if $q \in C_p$ in γ, then $l_p \in \text{PREFIXES}(l_q)$.

Lemma 11 *The system is self-stabilizing with respect to* Π_4.

Proof. By Lemma 10, for every process p, if C_p contains q such that $l_p \notin \text{PREFIXES}(l_q)$, then q is eventually removed from C_p. By Lemma 9, for every p, q can be added to C_p only if $l_p \in \text{PREFIXES}(l_q)$. So, eventually, if $q \in C_p$, then $l_p \in \text{PREFIXES}(l_q)$. □

It follows from Lemma 11 that, in every configuration γ satisfying Π_4, if there exists a process p and no q such that $l_p \in \text{PREFIXES}(l_q)$, then C_p is an empty set.

In other words, the leaf nodes will not have any children. We will now show that the number of trees will eventually become one.

Lemma 12 *In every execution starting from a configuration γ satisfying Π_4, the number of times a process p sets f_p to $\perp$ is less than or equal to* 1.

Proof. Assume by contradiction that there exists an execution e starting from γ and a process p setting f_p to $\perp$ more than once. In a configuration satisfying Π_4, by Corollary 5 and Lemma 7, p can set f_p to $\perp$ upon receipt of a message ORPHAN only. So, p receives ORPHAN at least twice. After the first receipt, p executes the loop Lines 2.01-2.29. There are two cases to consider:

1. $l_p = \epsilon$. In that case, p obtains an existing "ϵ-process" q' as its parent — refer to Lines 2.05-2.09. Then, p sends UPDATEPARENT to q' that will never sends ORPHAN to p since $l_q \in \text{PREFIXES}(l_p)$.

2. $l_p \neq \epsilon$. In that case, p creates and chooses as a parent a new "ϵ-process" q. This case is similar to the first one.

□

Let ϱ be the number of processes p such that $f_p = \perp$.

Lemma 13 *In every configuration γ satisfying Π_4, if $\varrho = 0$ in γ, then ϱ eventually becomes greater than* 0 *and remains greater than* 0 *thereafter.*

Proof. Assume by contradiction that $\varrho = 0$ in γ and there exists an execution e starting from γ such that ϱ is equal to 0 infinitely often. There are two cases to consider:

1. $\varrho = 0$ in every configuration of e, *i.e.*, $\forall p,\ f_p \neq \perp$ in every configuration. So, no process ever receives ORPHAN. Let p be a process such that $\forall q \neq p, l_q \notin \text{PREFIXES}(l_p)$—*i.e.*, l_p is minimum. (Note that in every configuration satisfying Π_4, $\forall q \neq p,\ p \notin C_q$.) Upon the first receipt of PARENT? sent by p to its parent, say p', p' sends ORPHAN to p. A contradiction.

2. $\varrho = 0$ infinitely often. From Lemma 12, $\forall p \in P$, p sets f_p at most once. So, ϱ increases from 0 to a value $x \leq |P|$. Then, since we assume that $\varrho = 0$ infinitely often, it means that ϱ will then be equal to 0, eventually. And since ϱ can not increase anymore, it will remains equal to 0, which is the first case.

□

Lemma 14 *In every execution starting from a configuration γ satisfying Π_4, ϱ eventually becomes equal to* 1.

Proof. By Lemmas 12 and 13, in every execution from γ, there exists a configuration γ_t such that ϱ is equal to a maximum value $x \in [1, |P|]$. Assume by contradiction that there exists an execution e, a value $y \in [2, x]$, and a configuration γ'_t in e with $t' \geq t$ such that $\varrho = y$ and remains equal to y thereafter. There are two cases to consider:

1. Among the y nodes, there exists p such that $l_p \neq \epsilon$. Then, p eventually executes Lines 2.11-2.13 a new ϵ-process is created, taking p as its child. The number of roots is unchanged but, eventually, every root is labeled by ϵ.

2. The label of the y nodes is equal to ϵ. Let p be the ϵ-processes having the maximum identifier. By executing Line 2.06, p eventually chooses an ϵ-process q such that q sets f_q to p upon receipt of the message UPDATEPARENT sent by p, and the number of roots is decremented. A contradiction.

□

Let Π_5 be the predicate over $\mathcal{C}$ such that $\varrho = 1$.

Lemma 15 *The system is self-stabilizing with respect to* Π_5.

Proof. Follows from Lemmas 12, 13, and 14. □

In every configuration satisfying Π_5, there exists a single process r such that $l_r = \epsilon$ and $f_r = \perp$. In the next and last step of the proof, we show that if the parent of a process p changes, then p moves toward the leaves such that the tree eventually forms a PGCP tree.

Lemma 16 *In every execution starting from a configuration γ satisfying Π_5, if a process p sets f_p to q, then $l_q \in \text{PREFIXES}(l_p)$.*

Proof. In every configuration γ satisfying Π_5, a process can change f_p by executing the receipt of either a message GRANDPARENT or UPDATEPARENT, in both cases, sent by its parent. In both cases, f_p is set to q such that $l_q \in \text{PREFIXES}(l_p)$. □

Lemma 17 *In every execution starting from a configuration γ satisfying Π_5, the number of pairs p, q such that $l_p = l_q$ eventually becomes equal to* 0.

Proof. Note that in every configuration γ satisfying Π_5, one among $\{p, q\}$ is the parent of the other. Without loss of generality, we assume that p is the parent of q. By the repeated executions of Lines 2.15-2.16 and 8.01-11.03 on each pair p, q, all the children of q eventually become the children p and q eventually disappears. □

Let Π_6 be the predicate over $\mathcal{C}$ such that $\gamma \in \mathcal{C}$ satisfies Π_6 iff the distributed data structure maintained by the variables of Algorithm 1 and 2 forms a Proper Greatest Common Prefix Tree. We want $\forall p, \forall q_1, q_2 \in C_p, l_p =GCP(l_{q_1}, l_{q_2})$. Considering the results of Lemmas 2, 7, 8, 11, 15, 16, and 17, there remains to eliminate problematic cases expressed by conditions of Line 2.17 and Line 2.21. By the repeated executions of Lines 2.17-2.27, we can claim the final result of our algorithm:

Theorem 18 *The system is self-stabilizing with respect to* Π_6.

5 Simulation results

The main goal of this section or running the simulation is to test and demonstrate the scalability of the proposed protocol. In particular, we investigated the convergence time and the number of messages exchanged both w.r.t. the number of nodes.

The simulator is written as a Python script. The script arbitrarily creates an initial faulty configuration of the network. By *arbitrarily*, we mean that each node is created independently from the others. To create one node, it picks a randomly created label (on the latin alphabet) of size between 1 and 20. It also chooses some nodes randomly from the set of already created nodes, to become the parent and the children of the node currently created. Thus, the initial graph is inconsistent — prefix relationship may be wrong (*e.g.,* a node label may be prefixed by the label of its child), or the information about the neighbors may be incorrect. For example, p may consider its parent label is l although q is labeled $l' \neq l$, or p may assume q as its parent while p does not consider q is its child. We created the tree randomly to test the power of the proposed self-stabilizing protocol.

The protocol is launched at each node of the graph. We assume a discrete time, and each period is a *processing sample*. In other words, one period begins when the first node starts the execution of the periodic rule, and the period ends when every node has triggered the periodic rule once and only once, and the set of actions resulting from it (sending messages, processing messages, updating variables, etc.) have been executed. As we detailed in the proof, this set is finite, since the maximal set of messages generated by one execution of the periodic rule is finite. To implement the discrete sampling, processes are *synchronized*. But, the discrete time reflects the slowest processor rate. In other words, the scheduler simulated is fair.

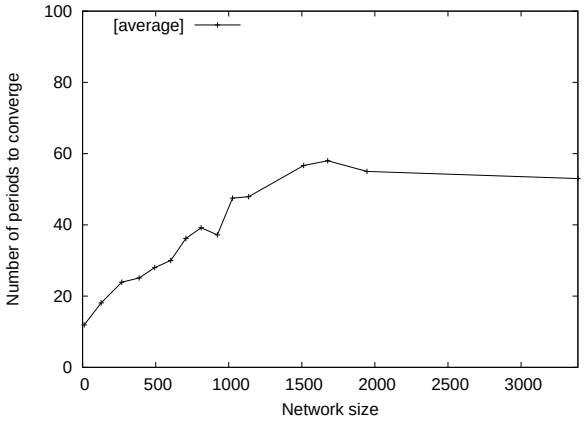

Figure 1. Simulation of the protocol: convergence time

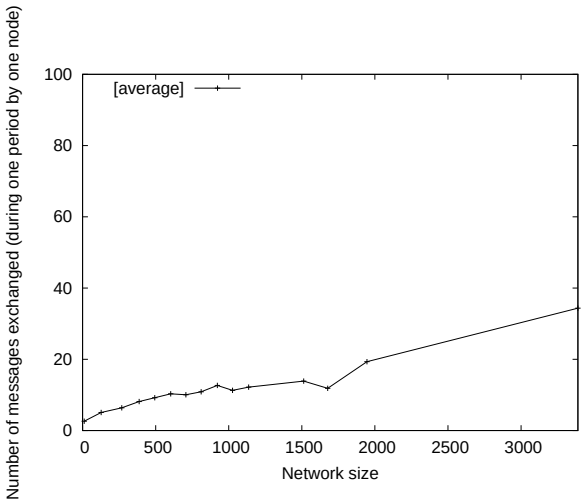

Figure 2. Simulation of the protocol: amount of messages

Figure 1 shows the number of periods required on an average to converge as a function of the number of the final number of nodes in the tree. Note that this number is equal to the initial number of distinct labels in the graph plus the number of labels created for the validity of the tree (in order to satisfy Definition 1). The plot in Figure 1 shows that the number of periods required to converge increases very slowly when the size of the

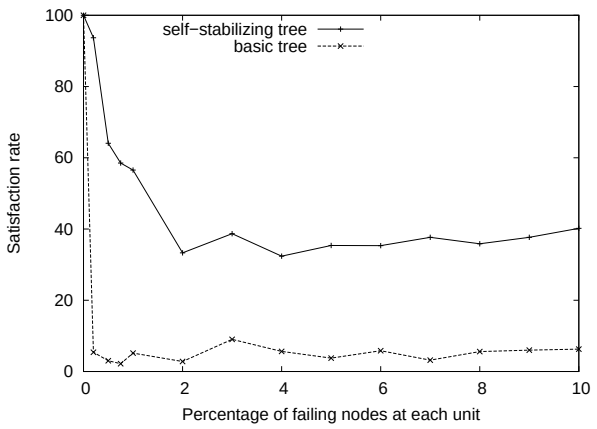

Figure 3. Simulation of the protocol: satisfaction rate of clients' requests

tree ranges from a couple of nodes to more than 3000. This suggests that the convergence time grows linearly in the worst case, and with a very low slope (approximately $1/50$). Figure 2 gives an average estimation of the number of messages each node exchanges during one period as a function of the final number of nodes. Again, the plot suggests a linear behavior in the worst case. These two results show that, when the tree grows, both the amount of processing and the number of messages exchanged by the nodes (and thus the utilization of CPU and network resources) grow slowly, indicating the scalability of the protocol.

Finally, we simulated clients' requests, *i.e.*, discovery requests looking up for a service. Discovery requests on a given service (or label) are encapsulated in a message sent to a randomly picked node. Then the message is routed until it reaches the node labeled by the requested service.

We investigated if a prefix tree overlay enhanced with our self-stabilizing protocol, regardless of the convergence time, allows the system to guarantee a certain level of availability. To this end, we simulated a prefix tree continuously undergoing failures, in a faster rate than the convergence time, under the same discrete-time conditions as the previous experiments. In Figure 3, the X-axis shows the number of nodes undergoing failures in percentage (0-10) of the total number nodes of the tree (about 500 in this experiment) at each period. The Y-axis gives the percentage of clients' requests satisfied. A request is said to be satisfied if it reached its destination in the tree starting from a random node. The curve shows that this number significantly improves when the self-stabilizing algorithm is used — approximately from 5% to 40% and in spite of very bad conditions, *i.e.*, 10% of nodes failing at each period. The *basic* tree includes no other fault-tolerance mechanism, like replication.

6 Conclusion

We presented a practical self-stabilizing protocol for the maintenance of a tree-structured P2P indexing system. While previous work in this area mainly relied on theoretical coarse grain models, this protocol used the message passing model, thus is implementable on P2P platforms. We provided a comprehensive and formal correctness proof of the proposed protocol. Simulations demonstrate that the convergence time and the communication of the protocol increases slowly when the tree grows, indicating a good scalability. Moreover, we showed the fault-tolerant features of the protocol indicating the improvement of the availability of service discovery systems.

We plan to implement the protocol inside a prototype of a peer-to-peer indexing system we are currently developing, based on the JXTA toolbox. Our initial experiments, conducted on the Grid'5000 platform [5], look promising.

References

[1] J. Aspnes and G. Shah. Skip Graphs. In *Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–393, January 2003.

[2] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilizing end-to-end communication. *Journal of High Speed Networks*, 5(4):365–381, 1996.

[3] A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *Proceedings of the SIGCOMM Symposium*, August 2004.

[4] A Bui, A K Datta, F Petit, and V Villain. Snap-stabilization and pif in tree networks. *Distributed Computing*, 20(1):3–19, 2007.

[5] 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.

[6] 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, *HiPC 2006*, 2006.

[7] E. Caron, F. Desprez, F. Petit, and C. Tedeschi. Snap-stabilizing Prefix Tree for Peer-to-peer Systems. In *SSS 2007*, pages 82–96. Springer Verlag Berlin Heidelberg, 2007.

[8] E. Caron, F. Desprez, and C. Tedeschi. A Dynamic Prefix Tree for the Service Discovery Within Large Scale Grids. In *P2P2006*. IEEE.

[9] 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.

[10] A K. Datta, M Gradinariu, M Raynal, and G Simon. Anonymous publish/subscribe in P2P networks. In *IPDPS'03. The 17th International Parallel and Distributed Processing Symposium*, page 74a, 2003.

[11] E. W. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Commun. ACM*, 17(11):643–644, 1974.

[12] S. Dolev. *Self-Stabilization*. The MIT Press, 2000.

[13] S Dolev and R I. Kat. Hypertree for self-stabilizing peer-to-peer systems. *to appear in Distributed Computing*, 2007.

[14] S. Dolev and J. L. Welch. Crash resilient communication in dynamic networks. *IEEE Transactions on Computers*, 46(1):14–26, 1997.

[15] B. Ducourthial and S. Tixeuil. Self-stabilization with path algebra. *Theoritical Computer Science*, 293(1):219–236, 2003.

[16] T. Herault, P. Lemarinier, O. Peres, L. Pilard, and J. Beauquier. A Model for Large Scale Self-Stabilization. In IEEE, editor, *21th International Parallel and Distributed Processing Symposium, IPDPS 2007*, 2007.

[17] 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.

[18] T. Herman and Masuzawa T. Available Stabilizing Heaps. *Information Processing Letters*, 77:115–121, 2001.

[19] A. Iamnitchi and I. Foster. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In *IPTPS*, pages 118–128, 2003.

[20] M. Cai and M. Frank and J. Chen and P. Szekely. MAAN: A multi-attribute addressable network for Grid information services. 2(1):3–14, March 2004.

[21] 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.

[22] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Adressable Network. In *ACM SIGCOMM*, 2001.

[23] 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.

[24] C. Schmidt and M. Parashar. Enabling Flexible Queries with Guarantees in P2P Systems. *IEEE Internet Computing*, 8(3):19–26, 2004.

[25] A Shaker and D S. Reeves. Self-stabilizing structured ring topology P2P systems. In IEEE, editor, *Fifth IEEE International Conference on Peer-to-Peer Computing, P2P 2005*, pages 39–46, 2005.

[26] 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.

[27] 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.

[28] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.