Self-Stabilizing Prefix Tree Based Overlay Networks

Eddy Caron, Ajoy K. Datta, Franck Petit, Cédric Tedeschi

Self-Stabilizing Prefix Tree Based Overlay Networks

Eddy Caron[1], Ajoy K. Datta[2], Franck Petit[3], and Cédric Tedeschi[4]

[1] LIP - UMR CNRS/ENS Lyon/UCB Lyon/INRIA 5668, University of Lyon, France

[2] School of Computer Science, University of Nevada Las Vegas, USA

[3] LIP6 UMR 7606, UPMC Sorbonne Universities, Paris, France

[4] University of Rennes 1/INRIA, France

December 19, 2019

Abstract

Computing over large platforms calls for the ability to maintain distributed structures at large scale. Among the many different structures proposed in this context, the prefix tree structure has been identified as an adequate one for indexing and retrieving information. One weakness of using such a distributed structure stands in its poor native fault tolerance, leading to the use of preventive costly mechanisms such as replication.

Self-stabilization is a suitable approach to design reliable solutions for dynamic systems, and was recently enhanced with new models to be able to deal with large scale dynamic platforms. A self-stabilizing system is guaranteed to reach a correct configuration, whatever its initial state is. Following this path, it is becoming possible to make distributed structures self-stabilizing at large scale.

In this paper, we focus on making tries self-stabilizing over such platforms, and propose a self-stabilizing maintenance algorithm for a prefix tree using a message passing model. The proof of self-stabilization is provided, and simulation results are given, to better capture its performances. Still based on simulations, we provide evidences that the protocol, beyond its capacity to repair the structure, can significantly improve the system's availability, even when the system is not yet stabilized.

Keywords: Distributed algorithms; Overlay Networks; Prefix Trees; Service Discovery; Fault-Tolerance; Self-Stabilization

1 Introduction

Platforms connecting geographically distributed computing resources have become a low cost alternative to supercomputers, bringing new challenges, related to their scale and dynamics. Information retrieval within such platforms, in particular to solve the resource discovery issue, was quickly identified as a big challenge, requiring fully-decentralized, or peer-to-peer (P2P) approaches [26]. The term "resource" should be here understood in a broad sense: it can refer to a storage facility, software utility or computing instruments.

The quest for the support of complex queries for resource discovery systems led to the design of different overlay structures. In particular, *tries*, whose variants are also known as *lexicographic trees*, *prefix trees*, and *radix trees*, have been identified as an adequate indexing structure. A trie is a particular tree in which nodes are labelled with strings and all descendants of a node are labelled with a string prefixed by the label of this node, the root node being usually labelled by the empty string. These structures allows to parallelize range queries, automated completion of partial search strings, and can be easily extended to support multi-attribute queries.

Although fault-tolerance is a mandatory feature in systems targeted for large scale platforms (in particular to avoid data loss and ensure a correct routing process), trie-based overlays offer a limited inherent robustness in a dynamic setting. The failure of one or more nodes can lead to the loss of the objects they store, and, more importantly, can lead to the inconsistencies in the distributed structure, making the system unable to process queries correctly. In recent trie-based overlay networks, fault tolerance was either ignored, or handled

through replication, which can be very costly in terms of computing and storage. Moreover, replication does not ensure the system's recovery after arbitrary transient failures affecting the memory, the computation, and the network.

In this paper, we take an alternate path to replication, and address the problem of the *maintenance* of a distributed trie. The solution proposed relies on self-stabilization. Self-stabilization [14, 15] is a general technique to design distributed systems that can handle arbitrary transient faults. A self-stabilizing system, regardless of the initial state of the processes and initial messages in the links, is guaranteed to converge to the intended behavior in finite time. The protocol designed to maintain a correct trie, referred to as the *Self-Stabilizing Trie (SST)* protocol, can tolerate any type of transient faults, in particular memory corruption and communication failures. Its correctness proof is provided. Simulation results are also provided, allowing to better capture its performance. In particular, we discuss the use of this approach in real settings through the simulation of the SST-based resource discovery system facing continuous failures. We show that using such a self-stabilizing approach can drastically improve the satisfaction ratio of queries sent by users, even during the repair phase.

Outline. The next section presents the series of works dealing with the maintenance of distributed structures in a dynamic setting. In particular, our work is put into perspective against recent works making distributed structures self-stabilizing. Section 3 presents the preliminaries for the algorithms, namely the models and the data structures considered. Section 4 presents the SST protocol, and gives a detailed correctness proof. Section 5 presents the simulation results. The contribution is summarized and open directions are discussed in Section 6.

2 Related Works

2.1 Trie-structured Overlay Networks

Information retrieval in P2P settings has been extensively studied. Early approaches were based on DHTs [36, 37, 40]. However, in spite of their scalability, their limit stands in their rather limited mechanisms of search, as they support only exact queries. A great deal of research went into finding ways to enhance them with more complex searching mechanisms. In particular, different algorithms were proposed to support multi-attribute range queries [5, 32, 38, 39]. In this research track, trie-structured overlays were introduced. Trie-based 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 [2, 12, 33, 35]. Let us briefly review some of them.

The prefix hash tree (PHT), proposed in [35], maintains a dynamic trie built with the set of possible keys, the trie being mapped over a DHT-structured network. Fault tolerance within PHT is delegated to the DHT layer. Skip Graphs, introduced in [2], are similar to tries, and build upon skip lists, using their own probabilistic fault-tolerance guarantees. P-Grid [12] is a similar binary trie whose nodes, in different sub-parts of the trie, are linked by shortcuts as in Kademlia [33]. Fault tolerance in P-Grid is based on probabilistic replication.

2.2 DLP-Tables

In this paper, we focus on the DLPT (Distributed Lexicographic Placement Table) approach, a trie-based information retrieval architecture, initially described in [8] in the context of service discovery over dynamic computational grids. The architecture is two-layered. The upper layer is a prefix tree indexing the information about services available on the platform. Each node of the tree maintains information about services sharing a particular name (or *key*), also used to label the tree node. This tree is built dynamically as services are registered by some servers in the computational platform, as illustrated by Figure 2.1. Nodes storing some services' references (and labelled by actual names of services) are grey-filled, the others, created to ensure the consistency of the index structure and enact the routing of queries, are labelled by *virtual* keys. In our

example, a DGEMM is first declared[1](a). A DTRSM is declared resulting in the creation of their parent, whose label is their greatest common prefix, *i.e.,* D (b). Finally, a DTRMM is added and the node DTR is created (c). This tree is mapped onto the lower layer, *i.e.,* the physical network. While this mapping is out of the scope of the paper, it may for instance rely on a distributed hash table to distribute the tree nodes onto processors. This issue has been devised in [9]. Simulations available in [8] show that, using different data sets, approximately 1/3 of nodes are labelled by *virtual* keys. In other words, 1/3 of the nodes simply ensures the consistency of the architecture, the others storing actual services' information. To avoid losing service information, it may be necessary to use replication. Still, ensuring the consistency of the structure and the routing process cannot be based on replication, which fails to address several requirements detailed above, and it requires new mechanisms. This paper proposes such mechanisms.

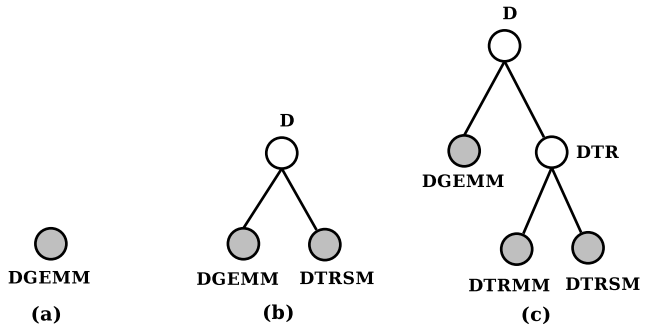


Figure 2.1: Construction of the prefix tree in DLPT approach.

To sum up, fault tolerance in trie-based overlay networks has been, until recently, mostly ignored, delegated to other layers, or implemented using replication mechanisms. In the following section, we detail the research efforts directed towards alternate, *best-effort* solutions to enhance these overlays with solutions to deal with the high dynamics of the platform.

2.3 Topology Maintenance in Dynamic Systems

Over the last years, a fair amount of works dealt with the maintenance of distributed structures facing the dynamics of the platform. Few works, like for instance [31] showed how to guarantee the continuous correctness of distributed structures under concurrent joins and leaves, but without considering failures.

Hayes *et al.* proposed the Forgiving Tree [22], a distributed structure which is ensured, under periodic adversarial deletion of one of its node, to maintain strong guarantees on its diameter and degree. Upon each adversarial deletion, both the time needed to recover and the number of messages sent are constant. The authors have extended their work to adversarial insertions of nodes in [21], but still with one adversarial move at a time. In [4], a spanning tree maintenance algorithm is proposed, also providing strong guarantees about degree and diameter, but under an infinite arrival model with bounded concurrency (the number of concurrently active nodes at a given time is bounded). However, these works still assume that processes are reliable.

Tackling the problem in faulty environments, some answers came from the self-stabilization area. Some investigations took interest in distributed search tree networks, *e.g.*, [24] and [25] for 2-3 trees and heap trees, respectively. A self-stabilizing lexicographic distributed structure is proposed in [19] as an application of r-operators. The protocol proposed in [7] deals with prefix tree maintenance. It has the nice property of being snap-stabilizing [6], *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,

[1]Labels used are function's names found in the BLAS linear algebra library [18]. For instance, DGEMM is a double-precision general matrix multiplication.

the algorithms in [7] requires the initial topology to be a rooted connected tree, and has been proved in a coarse theoretical model (namely, the state model introduced by Dijkstra in [14]).

However, we should here point out that all the previously mentioned solutions are designed for distributed systems defined by their topology, each node having a set of neighbors, and communicating with them through a finite number of links. In today's emerging platforms, like the Internet, each node $P1$ can communicate with any other node $P2$, provided that $P1$ knows the address of $P2$. The topology of P2P networks, or more generally of high-level protocols is logical, built on top of the physical network. Details of the physical topology, and the underlying routing process are abstracted. In other words, in a peer-to-peer overlay network, neighbors are the peers that a peer is aware of, *i.e.,* of which it knows the address. As a consequence, traditional models cannot be used to model peer-to-peer networks.

Nevertheless, new models have been proposed. Shaker and Reeves [1] give an intuitive and simple formalization of the bootstrapping problem. Recall that, to join the overlay, a node first needs to discover any node which is already a member of the overlay, by using an out-of-band mechanism. Shaker and Reeves put in words the fact that any peer-to-peer system needs a *weakly-connected* bootstrapping system, *i.e.,* able to gather the addresses of all alive nodes inside the overlay to ensure its convergence to a connected consistent overlay starting from any possibly disconnected topology. In the same paper, they give a self-stabilizing protocol to maintain an overlay network, assuming the presence of a continuous *weakly-connected* bootstrapping service. Each node periodically initiates lookup to the bootstrap system. Dolev and Kat [16] propose a similar *bootstrap-dependent* self-stabilizing overlay network based on their *HyperTree* structure. The HyperTree is a virtual tree structure built using IP addresses in which in-degree and out-degree of nodes are ensured to be $b \log_b n$ where n is the actual number of machines and b, an integer greater than one. The maximum number of hops in a lookup in the HyperTree is bounded by $\log_b n$. Following these two works, in [23], a model is proposed for the design of distributed algorithms for large scale systems, opening doors for further systematic investigation of self-stabilization solutions in peer-to-peer networks. A spanning tree maintenance protocol illustrates the model. Some works focused on the publish/subscribe paradigm, often implemented by peer-to-peer networks. Several papers, *e.g.,* [13, 41] design such protocols, but enhancing them with the self-stabilizing property.

A recent series of works proposed self-stabilizing overlay structures: In [11, 34], self-stabilizing protocol maintaining skip lists are proposed. In [27], a variant of the skip graph distributed structure is made self-stabilizing, with the main advantage of having a sublinear convergence time, which is not the case of previous attempts of self-stabilizing dynamic network structures. In [28], the same authors came up with a similar result (however, requiring polynomial time) for Delaunay graphs. Recently, they proposed a self-stabilizing version of Chord [29]. In [20], the authors introduced *topological* self-stabilization, which consists to guarantee that from any connected topology, an overlay with desirable properties eventually holds. They consider the local graph linearization, *i.e.,* how to build a distributed sorted list of the nodes in a self-stabilizing manner.

3 Preliminaries

In this section, we describe the distributed system model considered, specify the distributed data structures maintained, and formally present self-stabilization.

3.1 Model

A P2P network consists of a set of asynchronous processors, henceforth referred to as *peers*. By *asynchronous*, we mean that there is no bound on message delay, clock drift, or execution speed rate. Peers are endowed with distinct IDs. They communicate by exchanging messages. Any peer P_1 can communicate with another peer P_2 provided P_1 knows the *id* of P_2. We abstract out the details of the actual physical routing, as done in most peer-to-peer systems.

The distributed structure considered, used as an indexing system to store the information is a prefix tree. Nodes of this tree are mapped onto the peers of the network. Henceforth, the word *node* refers to a node of the tree, *i.e.,* a logical entity. Each peer maintains a part of the indexing system, *i.e.,* some (*logical*) nodes of the prefix tree. Each (*logical*) node is implemented as a *process* running on a peer. In other words, a

process *implements* a node. Each node is labelled. When the system is *correct* – such a state is defined later in Definition 3.1 – each node label is unique. We assume that each node keeps its label constant in a safe memory location, meaning that the labels are actual labels supposed to be stored within the trie. A single node is responsible for all services having a common name. However, initially, the system is not stabilized yet, and the structure may contain multiple nodes with 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 runs on all processes. *Nodes* and *processes* are basically different visions of the same object. In the remainder, we use these terms interchangeably. Recall that an important aspect of this work is that the topology itself can change during its reconstruction.

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 crashed or was deleted. The message delay is assumed to be bounded. We assume that messages are delivered in the order they were sent (channels are FIFO), and that, as long as a message is not processed by the receiving process, it is in transit.

We assume the presence of an *oracle* able to return process references. This oracle is similar to the one described in [23]. By calling it, a process can obtain the identifier of a random existing process of the system. In more detail, the oracle service provides two primitives. The first one, named GetRunningProcess(), returns the identifier of a randomly chosen process. The second one, named GetNewProcess() creates a new empty process (without initializing it) and returns the process' identifier. In order to be able to prove the termination of the algorithm, we need to assume that a finite number of calls to this service suffices to collect the identifiers of all processes in the system. The oracle may return the id of a process that left the system or never existed, but only during a limited amount of time. The implementation of such an oracle may rely on any centralized or decentralized directory, for instance based on a DHT enhanced with reliable broadcast [30]. A deeper discussion of this implementation falls beyond the scope of this paper.

3.2 Data Structure

We now formally describe the distributed structure we maintain. Let an ordered alphabet A be a finite set of letters. Let $\prec$ be an order on A. A non-empty *word* w over A is a finite sequence of letters $a_1, \ldots, a_i, \ldots, a_l$ where $l > 0$. 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, $w\varepsilon = \varepsilon w = w$. The *length* of a word w, denoted by $|w|$, is equal to the number of letters of w. $|\varepsilon| = 0$. 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 $w_1, \ldots, w_p$ ($p \geq 2$), denoted $GCP(w_1, \ldots, w_p)$ (resp. $PGCP(w_1, \ldots, w_p)$), is the longest prefix u shared by all of them (resp., with $\forall i$ such that $1 \leq i \leq p, u \neq w_i$).

Definition 3.1 (PGCP Tree) *A* Proper Greatest Common Prefix Tree *is a labelled 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 (itself excluded).*
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.*

In regard to this definition, configurations presenting different kinds of inconsistency can be envisioned, as illustrated by Figure 3.2. Figure 3.2(a) shows a correct PGCP tree, which is the final goal of the reconstruction. Figure 3.2(b) shows a *PGCP forest* comprising several disconnected PGCP trees. Such a situation can occur after some node left the system, as well as in the initial configuration. Other possible arbitrary initial configurations are shown in Figure 3.2(c) and 3.2(d). The former illustrates a labelled tree where nodes are arbitrarily placed. The latter combines a disconnection pattern, an arbitrary node placement, and a wrong topology (materialized by the presence of a cycle).

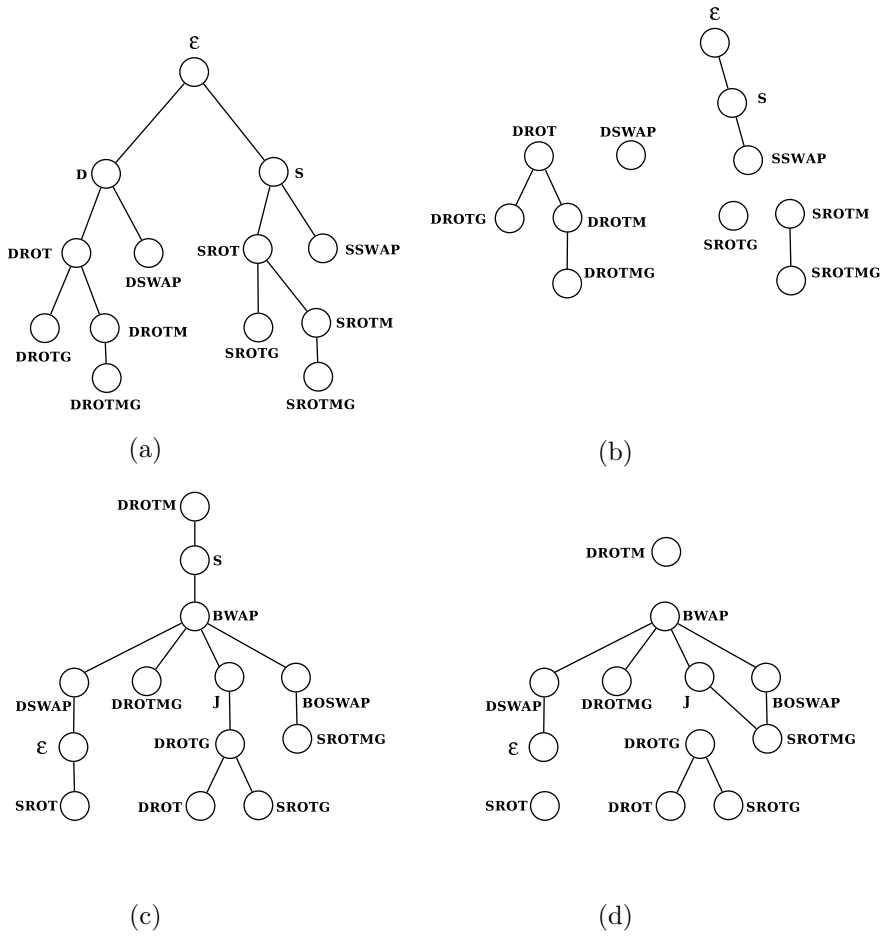


Figure 3.2: Possible configurations of the initial overlay.

3.3 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 over $\mathcal{C}$, 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.

Definition 3.2 (Self-Stabilization) *A transition system* $\mathcal{S} = (\mathcal{C}, \mapsto, \mathcal{I})$ *is a* self-stabilizing *system with respect to a predicate* Π *if and only if the following conditions hold:*

No Initialization: $\mathcal{I} = \mathcal{C}$,

Closure: Π *is closed,*

Convergence: *For every execution* e *of* $\mathcal{S}$*, there exists a configuration of* e *for which* Π *is true.*

4 Self-Stabilizing Trie Protocol

In this section, we present the SST protocol. This protocol is self-stabilizing and self-organizing, meaning, the initial state of the structure can be an arbitrary labelled graph, but will eventually converge to a PGCP tree. In particular, SST handles any configuration presented in Figure 3.2.

4.1 Assumptions

Communications. SST assumes the existence of an underlying *Self-Stabilizing End-to-End* (SSEE) communication protocol. Both layers communicate using **send/receive** primitives over FIFO message queues. The implementation of such an SSEE protocol falls beyond the scope of this paper. The reader may refer to [3, 17] for such protocols. Every process p (we use p to denote both the *id* of p and the address used by other processes to communicate with p) has a label l_p. We denote by $\widehat{p}$, the pair (p, l_p). Note that $\widehat{p} = \widehat{p'}$ is equivalent to $(p = p') \wedge (l_p = l_{p'})$. A 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 denoted $\widehat{C_p}$. Note that $\widehat{C_p}$, $\hat{p}$ and $\widehat{f_p}$ are variables, and that C_p is the result of a macro that extracts the first element of every pairs in $\widehat{C_p}$. The "**send**(<m>, q)" primitives sends message m to node q and returns a boolean. It always terminates; if the recipient q is alive, m is queued at q and *true* is returned; otherwise, (q is no longer available or m is lost) $false$ is returned.

Oracle Usage. The SST protocol assumes the function GetEpsilon(), which returns an arbitrary alive ε-process, *i.e.,* a node labelled by ε. Basically, the GetEpsilon() function calls the oracle primitive GetRunningProcess() as many times as necessary and checks whether the process returned is an ε-process. Since we assume that a finite number of calls to the oracle suffices to get all identifiers of alive nodes, GetEpsilon() also returns an ε-process in a finite time. The NewProcess(lbl, f, C) primitive initializes the parameters of a process. (The label, parent and children variables are filled with lbl, $\hat{f}$, and $\widehat{C}$, respectively.)

Failure Detection. We assume the presence of an underlying *heartbeat* protocol: any node that does not receive news from one of its children or parent during a given time, 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 protocol is made of a set of rules. The *periodic* rule runs on each node as detailed in Algorithm 1, periodically. The *upon receipt* rules, detailed in Algorithm 2, are initiated upon receipt of a message. All rules are atomic. (They can not be interrupted by the arrival of another message or by the expiration of the time out.)

4.2 Algorithm

The SST protocol is illustrated in Figure 4.3. It shows the steps involved in the repairing of a structure, where the periodic rule is first triggered on the node labelled $A - B$ (in steps *a*-*d*), and then on the node labelled ε_1 (in steps *e*-*f*).

The Prefixes(l) function returns all the strings which are a prefix of the string l, including l and ϵ. The GCP(l_1, l_2) function returns the greatest common prefix of strings l_1 and l_2.

Each node p periodically initiates the action described in Algorithm 1. p begins by eliminating the particular problematic cases where p is either a parent or a child of itself (Lines 1.03-1.04).

Lines 1.05-1.15 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 (refer to *d*-*e*), p tries to connect to another node q, also labelled ε. (On Figure 4.3(e), ε_1 discovers ε_2.) Then, q becomes a child of p (Line 1.09). p informs q that its parent changed using UpdateParent message. Upon receipt of that message, q updates its parent variable (Lines 5.01-5.02 of Algorithm 2). Since p and q are labelled identically, they will merge (the merging process is detailed below), thus reducing the number of roots by one.

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

1.02 upon TimeOut do
1.03     if f_p = p then f_p := ⊥ endif
1.04     if p ∈ C_p then Ĉ_p := Ĉ_p \ {p̂} endif
1.05     if f_p = ⊥ then
1.06         if l_p = ε then
1.07             q := GetEpsilon()
1.08             if q < p then
1.09                 Ĉ_p := Ĉ_p ∪ {(q, ε)}
1.10                 send(<UpdateParent, p̂>, q)
1.11             endif
1.12         else
1.13             new := GetNewProcess()
1.14             send(<Host, (ε, ⊥, {p̂})>, new)
1.15             f̂_p := (new, ε)
1.16         endif
1.17     endif
1.18     while ∃q ∈ C_p | l_q = l_p do
1.19         send(<Merge, p̂>, q)
1.20     done
1.21     while ∃(q_1, q_2) ∈ C_p^2 : l_p ∈ Prefixes(l_{q_1}) ∧ l_{q_1} ∈ Prefixes(l_{q_2})) do
1.22         send(<UpdateParent, q̂_1>, q_2)
1.23         Ĉ_p := Ĉ_p \ {q̂_2}
1.24     done
1.25     while ∃(q_1, q_2) ∈ C_p^2 : l_p ∈ Prefixes(l_{q_1}) ∧ l_p ∈ Prefixes(l_{q_2}) ∧ |GCP(l_{q_1}, l_{q_2})| > |l_p| do
1.26         l_new := GCP(l_{q_1}, l_{q_2})
1.27         new := GetNewProcess()
1.28         send(<Host, (l_new, p, {q_1, q_2})>, new)
1.29         send(<UpdateParent, new>, q_1)
1.30         send(<UpdateParent, new>, q_2)
1.31         Ĉ_p := Ĉ_p \ {q̂_1, q̂_2} ∪ {new}
1.32     done
1.33     if f_p ≠ ⊥ then
1.34         send(<Parent?, p̂>, f_p)
1.35     endif
1.36 done
```

2. If p is not labelled by ε, a new node labelled ε is artificially created as the parent of p. (See Figure 4.3($a-b$): the AB node creates the ε_2 node.) The new node ε_2 will eventually execute the periodic rule, and fall in the case explained before.

Lines 1.18-1.31 deal with children maintenance. They ensure that every set of children satisfies Definition 3.1, eventually. This phase eliminates sequentially, three families of problematic cases:

1. Firstly, we deal with cases where the set of children of p contains a node q whose label is the same as p. This is done by initiating the merging of node p and node q: On Line 1.19, p sends a MERGE message to such a process q. The steps involved upon receipt of the MERGE MESSAGE are given by Lines 7.01-10.03 in Algorithm 2: Upon receipt of the MERGE message, q informs its children that their new parent is their current grandparent (p is the parent of q), through GRANDPARENT messages. Upon receipt of this message, the children of q update their parent with p. To ensure a good synchronization, q waits until all of its children have been "adopted" by p, *i.e.,* q waits for the GFDONE message. q then informs p that the merging process is completed, through the sending the MDONE message. Finally, q terminates. This process is illustrated in Figure 4.3(f): ε_1 and ε_2 merge.

2. Secondly, we eliminate cases where a pair of children does not satisfy the second part of Definition 3.1. For instance, assume that a child of p, named q_1 prefixes another child q_2. The greatest common prefix of

Algorithm 2 *Upon receipt* rules, on process p

```
2.01  upon receipt of <PARENT?, q̂> do
2.02      if l_p ∈ PREFIXES(l_q) then
2.03          if send(<CHILD, p̂>, q) then
2.04              Ĉ_p := Ĉ_p ∪ {q̂}
2.05          endif
2.06      else
2.07          send(<ORPHAN, p̂>, q)
2.08          Ĉ_p := Ĉ_p \ {q̂}
2.09      endif
2.10  done

3.01  upon receipt of <CHILD, q̂> do
3.02      if f_p = q then l_{f_p} := l_q endif
3.03  done

4.01  upon receipt of <ORPHAN, q̂> do
4.02      if f_p = q then f_p := ⊥ endif
4.03  done

5.01  upon receipt of <UPDATEPARENT, q̂> do
5.02      if (l_q ∈ PREFIXES(l_p))∧ send(<PARENT?, p̂>, q) then f̂_p := q̂ endif
5.03  done

6.01  upon receipt of <HOST, l, f̂, Ĉ> do
6.02      NEWPROCESS (l, f̂, Ĉ)
6.03  done

7.01  upon receipt of <MERGE, q̂> do
7.02      if (f_p = q) ∧ (l_q = l_p) then
7.03          ∀q' ∈ C_p, send(<GRANDPARENT, f̂_p, p̂>, q')
7.04      endif
7.05  done

8.01  upon receipt of <GRANDPARENT, newf̂, q̂> do
8.02      if (f_p = q) ∧ (l_q ∈ PREFIXES(l_p)) then
8.03          f̂_p := newf̂
8.04          send(<GFDONE, p̂>, q)
8.05      endif
8.06  done

9.01  upon receipt of <GFDONE, q̂> do
9.02      if (q ∈ C_p) ∧ (l_p ∈ PREFIXES(l_q)) then
9.03          Ĉ_p := Ĉ_p \ {q̂}
9.04          if Ĉ_p = ∅ then
9.05              send(<MDONE, p̂>, f_p)
9.06              KILL()
9.07          endif
9.08      endif
9.09  done

10.01 upon receipt of <MDONE, q̂> do
10.02     if (q ∈ C_p) ∧ (l_p ∈ PREFIXES(l_q)) then
10.03         Ĉ_p := Ĉ_p \ {q̂}
10.04     endif
10.05 done
```

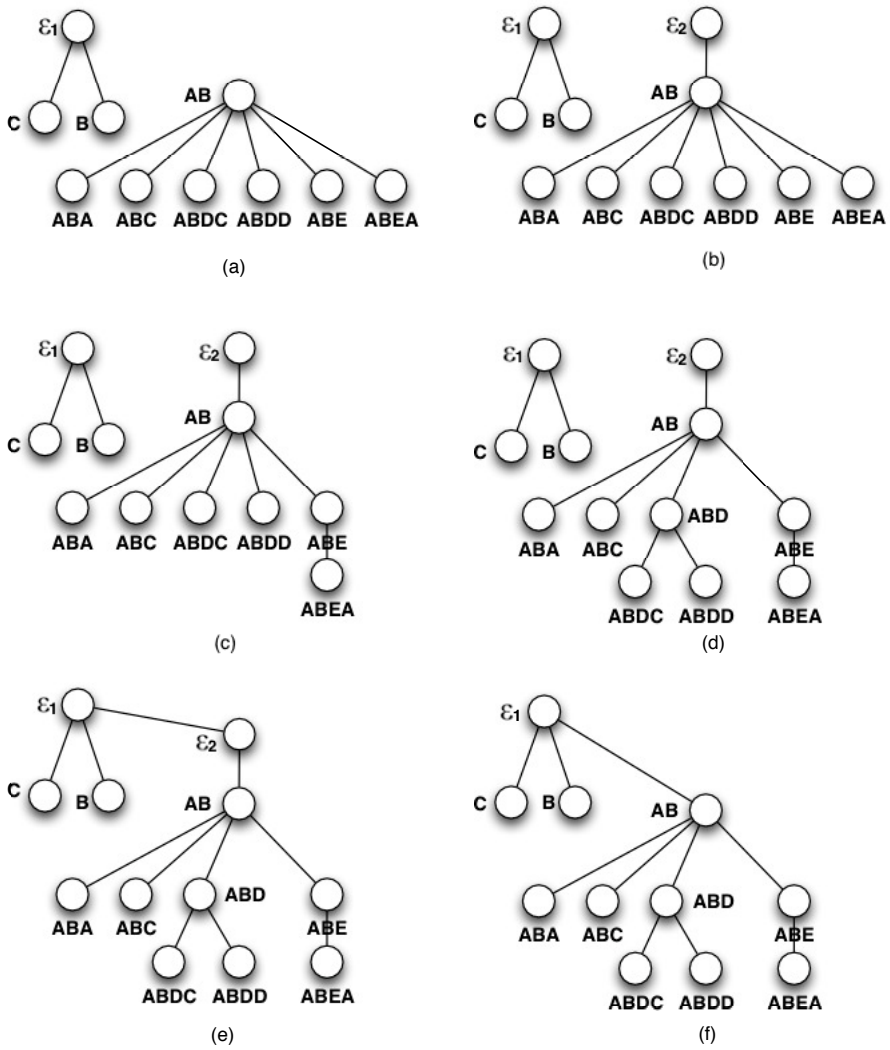


Figure 4.3: SST protocol

their labels is clearly equal to the label of q_1. However, their greatest common prefix, by Definition 3.1, must be the label of p. To repair this, q_2 becomes the child of q_1 (in Lines 1.21-1.23). On Figure 4.3(c), the $ABEA$ node becomes the child of the ABE node.

3. Thirdly, we check that there is no pair (q_1, q_2) in our set of children such that the greatest common prefix, say g, of their labels is greater than its own label (Lines 1.25-1.31). In this case, to satisfy Definition 3.1 (without deleting nodes), a new node must be created. This new node, labelled by g, is the child of p, and the common parent of q_1 and q_2. On Figure 4.3d, the ABD node is created.

The purpose of Lines 1.33-1.34 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 3.01-4.02 detail the action upon receipt of these messages. Upon receipt of CHILD, p updates the label of its parent. Upon receipt of ORPHAN, it becomes a root, and will execute the periodic rule, eventually.

4.3 Proof of Self-Stabilization

In this section, we show that, whatever its initial state is, the topology of the indexing system converges to a correct trie satisfying Definition 3.1 in a finite time, and that it remains correct as long as no new fault occurs. In order to prove the stabilization, we need to make two traditional assumptions: (i) The frequency of fault occurrence is low. (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 faults took place, *i.e.,* in this particular execution segment, no further faults will occur. Let P be the set of alive processes. Recall that 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 4.1 *The system is self-stabilizing with respect to* Π_1.

Corollary 4.1 *Every message received by a process in* P *in a configuration that satisfies* Π_1*, was sent by another process in* P.

According to Lemma 4.1, in the remainder of this paper, we consider executions starting from configurations satisfying Π_1 only.

Lemma 4.2 *Starting from a configuration satisfying* Π_1*, every process in* P *executes Lines 1.02-1.34 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 1.18-1.31) can loop forever. □

Corollary 4.2 *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 1.03 and 1.04 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 q such that $q \notin P$, *i.e., q* is alive.

Lemma 4.3 *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 4.4 *The system is self-stabilizing with respect to* Π_2.

Proof. From Corollary 4.1, 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 1.09 and 1.31 in which case q was returned by a GET*() function assumed to return ids in P. It can also add q using Line 2.04, in which case q was sent by q itself, and is thus alive. By Lemma 4.3, 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, according to Lemma 4.4, we consider only executions that start from configurations satisfying Π_2. Also, 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 1.02-1.34 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 4.2 and the fact that every message receipt action terminates.

Lemma 4.5 *The system is self-stabilizing with respect to* Π_3.

In the remainder, we consider that the executions start from configurations satisfying Π_3. Lemma 4.5 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 2.04 in which case the label was sent by q itself and is thus correct. Second, by using Line 1.09 in which case q was returned by GETEPSILON() and the label is set to ε. Third, by using Line 1.31 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_3, 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 4.6 *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 1.09. In this case, $l_p = l_q = \varepsilon$. (2) Line 1.31. 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 2.04. This line is executed only if $l_p \in \text{PREFIXES}(q)$ (Line 2.02). □

Lemma 4.7 *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 $l_p \notin \text{PREFIXES}(l_q)$. Two cases appear:

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 2.08). 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 4.8 *The system is self-stabilizing with respect to* Π_4.

Proof. By Lemma 4.7, 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 4.6, 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 $l_p \in \text{PREFIXES}(l_q)$. □

Corollary 4.3 *In every execution starting from a configuration γ satisfying Π_4, no process receives a message* ORPHAN *anymore.*

Proof. A message ORPHAN is sent from a parent to one of its child upon receipt of a message PARENT?. Since Π_4 holds, for every $q \in C_p$ in γ, l_p belongs to $\text{PREFIXES}(l_q)$. So, upon the receipt of a message PARENT? sent by q to p, p cannot send ORPHAN to q in a configuration γ that satisfies Π_4. □

We will now show that the number of trees will eventually become one. According to Lemma 4.8, we consider executions starting from configurations satisfying Π_4.

Lemma 4.9 *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 4.2 and Lemma 4.4, p can set f_p to $\perp$ upon receipt of a message ORPHAN only. So, p receives ORPHAN at least once, which contradicts Corollary 4.3. □

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

Lemma 4.10 *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 4.9, $\forall p \in P$, 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 4.11 *In every execution starting from a configuration γ satisfying Π_4, ϱ eventually becomes equal to* 1.

Proof. By Lemma 4.10, if $\varrho = 0$ in γ, then ϱ eventually becomes greater than 0 and remains greater than 0 thereafter. If there exists an execution that starts from γ during which ϱ increases, then by Lemma 4.9, then ϱ increases a finite number of times bounded by $|P|$. Therefore, in every execution from γ, there exists a configuration γ_t from which ϱ reaches 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 $\gamma_{t'}$ 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 $l_p \neq \varepsilon$. Then, p eventually executes Lines 1.13-1.15 a new ε-process is created, taking p as its child. The number of roots is unchanged but, eventually, every root is labelled 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 1.07, 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 4.12 *The system is self-stabilizing with respect to Π_5.*

Proof. Follows from Lemmas 4.9, 4.10, and 4.11. □

In every configuration satisfying Π_5, there exists a single process r such that $l_r = \varepsilon$ 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. We consider only executions that start from configuration satisfying Π_5.

Lemma 4.13 *In every execution starting from a configuration γ satisfying Π_5, if a process p sets f_p to q, then $l_q \in$ 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 $l_q \in$ PREFIXES(l_p). □

Lemma 4.14 *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 1.18-1.19 and 7.01-10.03 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 4.1, 4.4, 4.5, 4.8, 4.12, 4.13, and 4.14, there remains to eliminate problematic cases expressed by conditions of Line 1.21 and Line 1.25. By the repeated executions of Lines 1.21-1.31, we can claim the final result of our algorithm:

Theorem 4.1 *The system is self-stabilizing with respect to Π_6.*

5 Simulation

Our work is in line with the progress of computing platforms from centralized control and static settings toward *autonomic* distributed infrastructures, *i.e.,* bringing *self-** capabilities to the system, mainly self-management and self-adaptiveness. Self-stabilization turns out to be a suitable approach to reach this goal. In this section, we aim at highlighting such approach toward implementation in real environment. We also provide a rough complexity of convergence time in an environment close to the intended platforms.

In order to do so, we establish the convergence time in terms of *rounds* that is a well-known and widely used time complexity in the area of self-stabilization [15]. The notion of round has been defined in a model where the communications are abstracted by the ability for neighboring nodes to share registers. It captures the execution rate of the slowest node in any execution. In the message passing model, we get close to this concept by considering that both the message transit time and the computation time are globally constant. More specifically, the simulator was implemented using Python and implements discrete time: at each time step i, a message sent at step $i-1$ is received by the destination node, which, on receipt, triggers the action to be triggered upon receipt of the message (possibly generating new messages, in their turn to be received at step $i+1$).

5.1 SST Scalability

We tested the scalability of the protocol facing a randomly created initial configuration. The script developed creates this initial configuration. More precisely, a high level of randomness is used, ensuring the topology created suffers from many problems with high probability (*w.h.p.*). In the following, *random* refers to *uniformly random*. Nodes are constructed sequentially. Firstly, to create one node, a label is constructed, with a size randomly chosen in the range $[1, 20]$, and its composing characters are randomly picked in the Latin alphabet. Secondly, some nodes are randomly chosen from the set of already created nodes, to be its parent and children, in the sense that this node considers them as parent and children, but this relation is not symmetric *w.h.p.*, since these last nodes have also chosen their relatives in a random manner. For example, p may consider its parent label is l although q is labelled $l' \neq l$, or p may assume q as its parent while q does not consider p as its child. Doing so, the initial graphs contain inconsistencies of different nature (*w.h.p.*), related to the prefixing rules, to the neighboring symmetry, and, more simply, to the copy of the labels of one's neighbors.

As illustrated in Figure 5.4(a), when the size of the graph increases, the convergence time increases logarithmically. To explain this, we need to observe that during the reconstruction, two phases appear. The first one, dealing with the non-symmetric connections and the inconsistent labels. As these actions can be achieved with a lot of parallelism on distinct sets of nodes, they globally require a constant number of steps. The second phase consists in nodes aggregating into trees. The operation of merging two trees has a complexity related to their depths.

Figure 5.4(b) gives the average number of messages each node exchanges during one period, as a function of the number of nodes. It suggests a linear behavior in the worst case, with a very low slope (≈ 0.08).

These two results confirm that the utilization of the CPU and network resources grow slowly, when the size of initial graph increase.

5.2 SST in Practice

Finally, we wanted to explore the advantage of using such an approach in real settings, by observing the satisfaction ratio of discovery requests sent to a structure continuously undergoing failure, but enhanced with the SST protocol running in background.

To do so, discovery requests on a given service (or label) are encapsulated into a message, which is sent to a randomly picked node. Then, the message is routed in the structure until it reaches the node labelled by the name of the requested service. The question to be investigated is then "Whatever the actual state of the structure is (correct, or on its way to be correct), to what extent the SST protocol allows to improve the satisfaction of requests sent ?" Thus the tree simulated continuously underwent failures, at a rate higher than the convergence time, making the tree continuously incorrect, under the same discrete-time conditions as used before.

In Figure 5.5, the X-axis gives the time (60 discrete steps). The tree size is approximately 70. In this case, as we can see on previously discussed Figure 5.4(a), slightly more than 10 time steps are needed to converge. So, in this experiment, some *faults* are injected (inconsistant labels or connection) every 10 steps. At each step, a set of requests are sent to the tree. The Y-axis gives the percentage of clients' requests satisfied. A request is considered as satisfied if it reached its destination node in the tree, whatever the node initially contacted by the request is. Figure 5.5(a) shows the results when the network does not provide any fault-tolerance mechanism, and Figure 5.5(b) shows the results when the system is enhanced with the SST protocol.

As it can be expected for the non fault-tolerant version of the system, the satisfaction ratio drops quickly, as soon as some failures are injected. Moreover, each new set of failures injected affects more the satisfaction ratio. In contrast, when the system is enhanced with the SST protocol, even if the tree is never fully correct, we observe that the satisfaction ratio is greatly improved as soon as the protocol starts handling errors introduced: The ratio is almost doubled after one step. This suggests, beyond the actual convergence to the correct topology, that the algorithm allows to reach a *good* state quickly. This stands in the fact that the algorithm is highly parallel, making it possible to perform a lot of repairing action in a short time.

6 Conclusion

This paper provides a self-stabilizing distributed protocol able to recover a correct trie starting from an arbitrary labelled graph topology, with nodes in an arbitrary state, and links containing arbitrary messages.

Simulations demonstrated that the convergence time follows a logarithmic behavior and that the amount of communications induced grows slowly with the size of the tree. We also conducted simulations about a more practical impact of such a protocol. They suggest that the SST protocol, beyond its formal convergence, can significantly improve the performance of a prefix-tree based distributed indexing system, even in a highly dynamic setting.

Recently, the SST protocol was implemented within the service discovery component of SBAM[2], a decentralized middleware system, and deployed over a real platform [10]. Those experiments confirmed the behavior presented in Section 5.

[2] `http://graal.ens-lyon.fr/SBAM`

References

[1] A. Shaker and D S. Reeves, Self-Stabilizing Structured Ring Topology P2P Systems, *Fifth IEEE International Conference on Peer-to-Peer Computing, P2P 2005*, ed. IEEE (2005), pp. 39–46.

[2] J. Aspnes and G. Shah, Skip Graphs, *Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, (January 2003).

[3] B. Awerbuch, B. Patt-Shamir and G. Varghese, Self-Stabilizing End-to-End Communication, *Journal of High Speed Networks* **5**(4) (1996) 365–381.

[4] T. Bansal and N. Mittal, A scalable algorithm for maintaining perpetual system connectivity in dynamic distributed systems, *IPDPS*, (2010), pp. 1–12.

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

[6] A. Bui, A. K. Datta, F. Petit and V. Villain, Snap-Stabilization and PIF in Tree Networks, *Distributed Computing* **20(1)** (2007) 3–19.

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

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

[9] E. Caron, F. Desprez and C. Tedeschi, Efficiency of Tree-Structured Peer-to-Peer Service Discovery Systems, *5th International Workshop on Hot Topics in Peer-to-Peer Systems, Hot-P2P 2008, in conjunction with IPDPS*, (IEEE, Miami, USA, April, 14-18 2008).

[10] E. Caron, F. Chuffart and C. Tedeschi, When self-stabilization meets real platforms: An experimental study of a peer-to-peer service discovery system, *Future Generation Comp. Syst.* **29**(6) (2013) 1533–1543.

[11] T. Clouser, M. Nesterenko and C. Scheideler, Tiara: A self-stabilizing deterministic skip list, *SSS*, (2008), pp. 124–140.

[12] A. Datta, M. Hauswirth, R. John, R. Schmidt and K. Aberer, Range Queries in Trie-Structured Overlays, *The Fifth IEEE International Conference on Peer-to-Peer Computing*, (2005).

[13] A. K. Datta, M. Gradinariu, M. Raynal and G. Simon, Anonymous Publish/Subscribe in P2P Networks, *IPDPS'03. The 17th International Parallel and Distributed Processing Symposium*, (2003), p. 74a.

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

[15] S. Dolev, *Self-Stabilization* (The MIT Press, 2000).

[16] S. Dolev and R. I. Kat, HyperTree for Self-Stabilizing Peer-to-Peer Systems, *Distributed Computing* **20**(5) (2008) 375–388.

[17] S. Dolev and J. L. Welch, Crash Resilient Communication in Dynamic Networks, *IEEE Transactions on Computers* **46**(1) (1997) 14–26.

[18] J. J. Dongarra, J. D. Croz, S. Hammarling and I. S. Duff, A set of level 3 basic linear algebra subprograms, *ACM Trans. Math. Softw.* **16**(1) (1990) 1–17.

[19] B. Ducourthial and S. Tixeuil, Self-Stabilization with Path Algebra, *Theoritical Computer Science* **293**(1) (2003).

[20] D. Gall, R. Jacob, A. W. Richa, C. Scheideler, S. Schmid and H. Täubig, A note on the parallel runtime of self-stabilizing graph linearization, *Theory Comput. Syst.* **55**(1) (2014) 110–135.

[21] T. P. Hayes, J. Saia and A. Trehan, The forgiving graph: a distributed data structure for low stretch under adversarial attack, *PODC*, (2009), pp. 121–130.

[22] T. Hayes, N. Rustagi, J. Saia and A. Trehan, The forgiving tree: a self-healing distributed data structure, *PODC*, (2008), pp. 203–212.

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

[24] T. Herman and T. Masuzawa, A Stabilizing Search Tree with Availability Properties, *Proceedings of the 5th International Symposium on Autonomous Decentralized Systems (ISADS'01)*, ed. IEEE (2001), pp. 398–405.

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

[26] A. Iamnitchi and I. Foster, On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing., *IPTPS*, (2003), pp. 118–128.

[27] R. Jacob, A. W. Richa, C. Scheideler, S. Schmid and H. Täubig, A distributed polylogarithmic time algorithm for self-stabilizing skip graphs, *PODC*, (2009), pp. 131–140.

[28] R. Jacob, S. Ritscher, C. Scheideler and S. Schmid, A self-stabilizing and local delaunay graph construction, *ISAAC*, (2009), pp. 771–780.

[29] S. Kniesburges, A. Koutsopoulos and C. Scheideler, Re-chord: A self-stabilizing chord overlay network, *Theory Comput. Syst.* **55**(3) (2014) 591–612.

[30] J. Li, K. Sollins and D. Lim, Implementing Aggregation and Broadcast over Distributed Hash Tables, *SIGCOMM Comput. Commun. Rev.* **35**(1) (2005) 81–92.

[31] X. Li, J. Misra and C. G. Plaxton, Concurrent maintenance of rings, *Distributed Computing* **19**(2) (2006) 126–148.

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

[33] P. Maymounkov and D. Mazieres, Kademlia: A Peer-to-Peer Information System Based on the XOR Metric, *Proceedings of IPTPS02*, Cambridge, USA (March 2002).

[34] R. M. Nor, M. Nesterenko and C. Scheideler, Corona: A stabilizing deterministic message-passing skip list, *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, *SSS'11* (2011), pp. 356–370.

[35] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein and S. Shenker, Prefix Hash Tree: an Indexing Data Structure over Distributed Hash Tables, *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, (2004).

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

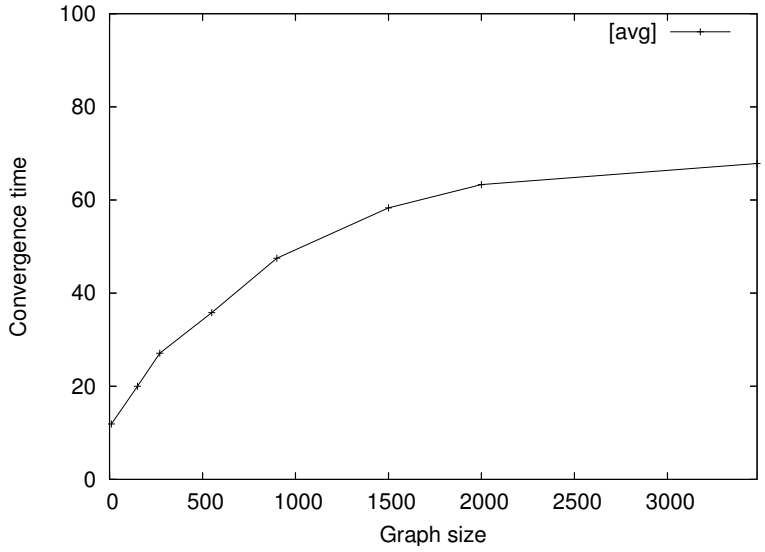
[37] A. Rowstron and P. Druschel, Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-To-Peer Systems, *International Conference on Distributed Systems Platforms (Middleware)*, (November 2001).

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

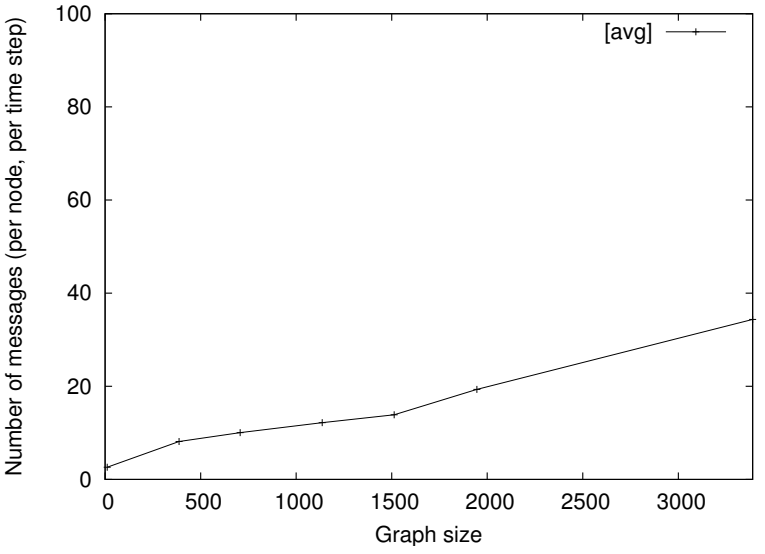
[39] Y. Shu, B. C. Ooi, K. Tan and A. Zhou, Supporting Multi-Dimensional Range Queries in Peer-to-Peer Systems., *Peer-to-Peer Computing*, (2005), pp. 173–180.

[40] I. Stoica, R. Morris, D. Karger, M. Kaashoek and H. Balakrishnan, Chord: A Scalable Peer-to-Peer Lookup service for Internet Applications., *ACM SIGCOMM*, (2001), pp. 149–160.

[41] Z. Xu and P. K. Srimani, Self-Stabilizing Publish/Subscribe Protocol for P2P Networks, *7th international workshop on Distributed Computing (IWDC 2005)*, ed. S. LNCS 3741 (2005), pp. 129–140.

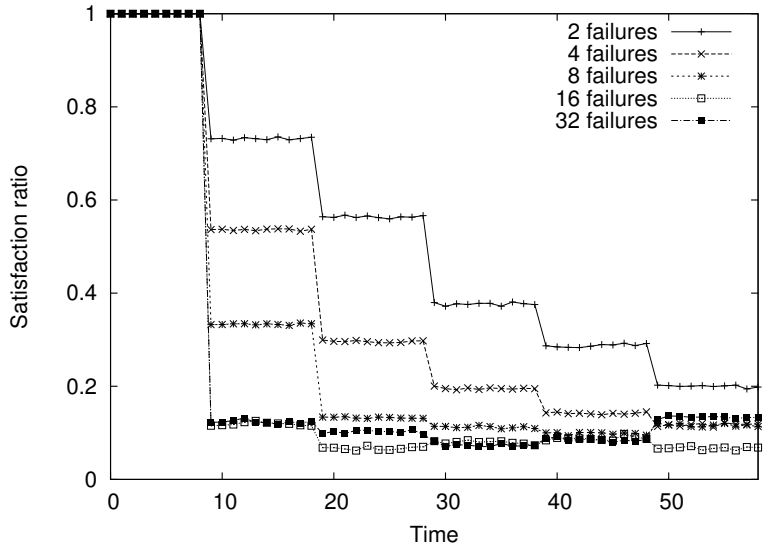


(a) Convergence time.

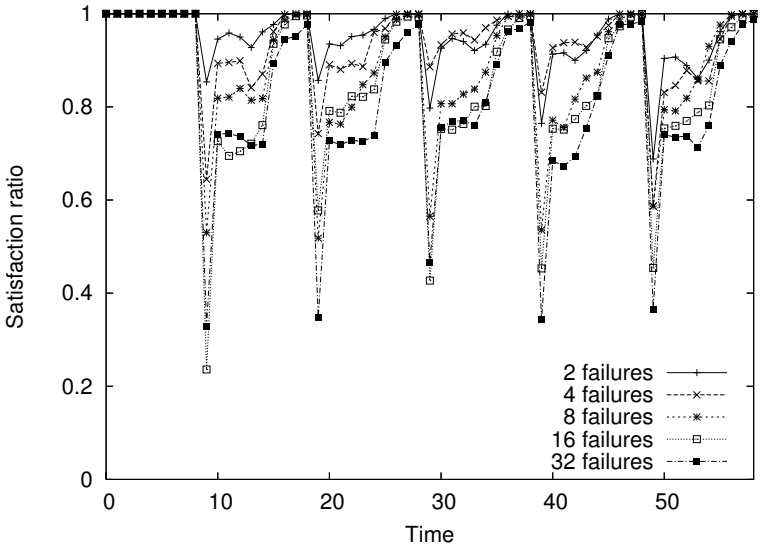


(b) Communication amount.

Figure 5.4: SST protocol: Repairing arbitrary topologies.



(a) Basic network.



(b) SST-enhanced network.

Figure 5.5: The SST protocol facing continuous faults.