



# A Self-stabilizing Hashed Patricia Trie

Till Knollmann<sup>1</sup>(✉) and Christian Scheideler<sup>2</sup>(✉)

<sup>1</sup> Heinz Nixdorf Institute, Computer Science Department, Paderborn University,  
Paderborn, Germany

tillk@mail.upb.de

<sup>2</sup> Computer Science Department, Paderborn University, Paderborn, Germany  
scheideler@upb.de

<https://www.hni.uni-paderborn.de/alg/>

<https://cs.uni-paderborn.de/ti/>

**Abstract.** While a lot of research in distributed computing has covered solutions for self-stabilizing computing and topologies, there is far less work on self-stabilization for distributed data structures. Considering crashing peers in peer-to-peer networks, it should not be taken for granted that a distributed data structure remains intact. In this work, we present a self-stabilizing protocol for a distributed data structure called the *hashed Patricia Trie* (Kniesburges and Scheideler WALCOM'11) that enables efficient prefix search on a set of keys. The data structure has a wide area of applications including string matching problems while offering low overhead and efficient operations when embedded on top of a distributed hash table. Especially, longest prefix matching for  $x$  can be done in  $\mathcal{O}(\log |x|)$  hash table read accesses. We show how to maintain the structure in a self-stabilizing way. Our protocol assures low overhead in a legal state and a total (asymptotically optimal) memory demand of  $\Theta(d)$  bits, where  $d$  is the number of bits needed for storing all keys.

**Keywords:** Self-stabilizing · Prefix search · Distributed data structure

## 1 Introduction

We consider the problem of maintaining a distributed data structure for efficient *Longest Prefix Matching* in peer-to-peer (P2P) systems. We focus on the *hashed Patricia Trie* (HPT) introduced in [14] and present an algorithm rendering a self-stabilizing version of this data structure when applied on top of any reliable *distributed hash table* (DHT).

**Definition 1 (Longest Prefix Matching).** Consider a set of binary strings called keys and a binary string  $x$ . The task of *Longest Prefix Matching* is to find a key  $y$  sharing the longest common prefix with  $x$ . A *prefix* of a binary string is a substring beginning with the first bit. We denote the longest common prefix of  $x$  and  $y$  by  $\text{lcp}(x, y)$ .

---

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Center ‘On-The-Fly Computing’ (SFB 901).

We denote a prefix  $p$  of  $x$  by  $p \sqsubseteq x$ .  $p$  is a *proper prefix* of  $x$  ( $p \sqsubset x$ ) if  $p$  is a prefix of  $x$  and  $|p| < |x|$ , where  $|p|$  is the length of  $p$ . Longest Prefix Matching is an old problem with applications in various areas including string matching problems and IP lookup in Internet routers. To solve it efficiently in a distributed P2P system, the HPT has been introduced [14]. The HPT is a distributed data structure applied to any common DHT which allows efficient prefix search for  $x$  in  $\mathcal{O}(\log |x|)$  read accesses to the hash table, i.e., solely based on the length of the search word  $x$ . The costs for an insertion of  $x$  is in  $\mathcal{O}(\log |x|)$  read accesses and  $\mathcal{O}(1)$  write accesses, while deletion can be done in  $\mathcal{O}(1)$  accesses. The memory space used is asymptotically optimal in  $\Theta(\text{sum of all key lengths})$ . Moreover, *Suffix Trees* can be implemented efficiently using Patricia Tries and thus also hashed Patricia Tries (called *PAT Trees* [10]). This allows us to efficiently decide if a given string  $x$  is a substring of a text in a runtime only depending on the length of  $x$ .

The usefulness of Patricia Tries motivates us to investigate how a HPT can be maintained in a P2P system where nodes may enter/leave or even fail. While a lot of research has considered the design of self-stabilizing computation or topologies (see Sect. 1.2), to the best of our knowledge there are far fewer results concerning self-stabilizing distributed data structures. However, failures of peers may affect the correctness of any distributed data structure. Therefore, we consider the problem of finding an efficient distributed protocol to maintain a HPT in a self-stabilizing way.

## 1.1 Model

We assume the existence of a self-stabilizing *distributed hash table* (DHT) which provides the operations  $\text{DHT-INSERT}(x)$  to insert data and  $\text{DHT-SEARCH}(x)$  to retrieve data. These operations are carried out reliably on the stored data, i.e., no operation is ever canceled. We assume the existence of a collision-free hash function which maps binary strings to positions in  $[0, 1)$  to store data in the DHT. The function is available locally at every peer. Each peer has a unique identifier, manages local variables and maintains a *channel*. When a peer sends a message  $m$  to peer  $p$ , it puts  $m$  in the channel of  $p$ . A channel has unbounded capacity and messages never get lost. If a peer processes a message in its channel, the message is removed from the channel afterwards.

We distinguish between two types of *actions*: The first one is for standard procedures and has the form  $\langle \text{label} \rangle (\langle \text{parameters} \rangle) : \langle \text{command} \rangle$  where *label* is the name of the action, *parameters* define the set of parameters and *command* defines the statements that are executed when calling the action. It may be executed locally or remotely. The second type has the form  $\langle \text{label} \rangle : (\langle \text{guard} \rangle) \rightarrow \langle \text{command} \rangle$  where *label* and *command* are defined as above and *guard* is a predicate over local variables. An action at peer  $p$  can only be executed if its guard is *true* or a message in the channel of  $p$  requests to call it. We call such an action *enabled*. The guard of our protocol routine `TIMEOUT` is always *true*.

A *state* of the system is defined by the assignment of variables at every peer, the data items and their values stored at every peer and all messages in

channels of peers. The system can transform from a state  $s$  to another state  $s'$  by executing an enabled *action* at a peer. An infinite sequence of states  $(s_1, s_2, \dots)$  is a *computation* if  $s_{i+1}$  can be reached by executing an action enabled in  $s_i$  for all  $i \geq 1$ . The state  $s_1$  is called *initial state*. We assume *fair message receipt*, i.e., every message contained in a channel is eventually processed. Also, we assume *weakly fair action execution* such that any action that is enabled in all but finitely many states is executed infinitely often. This especially applies to the TIMEOUT procedure. We call a protocol *self-stabilizing* if it fulfills *convergence* and *closure*. Convergence means that starting from an arbitrary initial state, the protocol transforms the system to a legal state in finite time. Closure means that starting from a legal state, the protocol only transforms the system to consecutive legal states. Our goal is to provide a self-stabilizing HPT. We define the legal state of a HPT later in Sect. 4.1.

## 1.2 Related Work

The basic data structure we consider here is the Patricia Trie. This compressed tree structure has been introduced by Morrison in [16]. It was extended to the hashed Patricia Trie by Kniesburges and Scheideler in [14]. In [10], Gonnet et al. presented PAT Trees which are essentially Patricia Tries for special suffixes (*sistrings*) of a text. This widens the applications of Patricia Tries to general string problems such as deciding if a word or sentence is contained in a text [10]. The work on self-stabilization started with the research of Dijkstra in [7] where he analyzed self-stabilization in a token ring scenario. Since then, research has covered wide areas including self-stabilizing computation [3, 5] and coordination [1, 2, 7, 9]. Furthermore, with the rise of P2P systems [18, 20], self-stabilizing topologies in the sense of overlay networks gained attraction [4, 6, 8, 11–13, 19]. We use approaches originally presented for topological self-stabilization. This includes a technique called *Linearization* presented by Onus et al. in [17]. A common approach for storing data in overlay networks is a distributed hash table (DHT) like Chord [20]. Using hashing, data items, as well as network peers, are mapped to the  $[0, 1)$  interval such that a mapping between them is established. There are various results on self-stabilizing DHTs in the literature (for example [13]). Further, most (self-stabilizing) overlay networks can easily be extended to a DHT given sortable unique identifiers for the peers which is a common assumption.

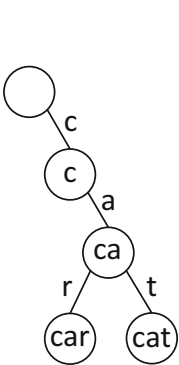
## 1.3 Our Contribution

We present a self-stabilizing protocol called SHPT to maintain a slightly modified version of the HPT as presented in [14]. Whenever we refer to HPT, we implicitly mean the modified version. The HPT and our modification are briefly introduced in Sect. 2. Afterwards, Sect. 3 gives a high-level description of the most important mechanisms of our protocol. We only require for an initial state that the underlying DHT is in a legal state and that a set of unique keys is stored at DHT nodes. In Sect. 4, we show that our protocol stabilizes a HPT in finite

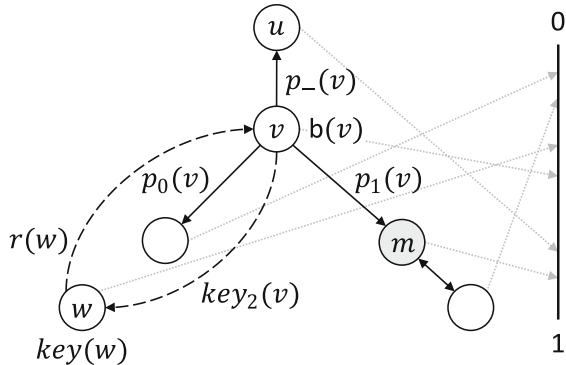
time out of any initial state. When the HPT is in a legal state, our protocol guarantees a low overhead of a constant amount of hash table read accesses and messages generated at each DHT node per call of the protocol routine. Furthermore, we can bound the total memory consumption in a legal state to  $\Theta(d)$  bits if  $d$  is the number of bits needed to store all keys. Due to space limitations, we deferred the Pseudocode and the full proofs concerning correctness and overhead to the full version [15].

## 2 Hashed Patricia Trie

We consider a data structure called the *hashed Patricia Trie* (HPT) as presented in [14]. The HPT is an extended Patricia Trie that is distributed in a P2P System by using a DHT. We briefly describe the construction. For details, we refer to [14]. The Patricia Trie is a compressed trie which was proposed by Morrison in [16]. Suppose we are given a key set KEYS consisting of strings. A trie is a tree structure that consists of labeled nodes and labeled edges. The root node is labeled by the empty string and every edge is labeled by one character. The label of a node is the concatenation of all edge labels of edges traversed on the unique path from the root to the node. For each  $k \in \text{KEYS}$  there is a node labeled by  $k$  (see Fig. 1). The Patricia Trie introduces compression by allowing edge labels to be strings such that inner nodes with a single child, which do not represent a key, can be avoided. Similar to [14], we restrict ourselves to keys represented by binary strings. We store the Patricia Trie in a DHT by hashing all nodes by their label resulting in the hashed Patricia Trie. Our notation is close to the one of [14] and can be seen in Fig. 2.



**Fig. 1.** Example of a classical Trie containing the keys “car” and “cat”.



**Fig. 2.** Values stored at nodes of the HPT from the perspective of  $v$ . Nodes are stored by hashing their label to  $[0, 1)$  in combination with a DHT. White nodes denote Patricia nodes while Msd nodes are depicted in gray.

Every Patricia node  $v$  has a label denoted by  $b(v)$  and stores three edges. The *root* node stores the empty string  $b(\text{root}) = \varepsilon$ .  $p_-(v)$  is the parent edge of

$v$  pointing to the parent node  $u$  such that  $b(u) \circ p_-(v) = b(v)$ . We denote by  $\circ$  the concatenation of strings. By  $p_x(v)$  we denote the child edge of  $v$  starting with the value  $x$  for  $x \in \{0, 1\}$ . If  $b(w) \in \text{KEYS}$  for a Patricia node  $w$ , we set  $\text{key}(w) = b(w)$ . Additionally, an inner Patricia node stores a  $\text{key}_2(v) = k$ , where  $k$  is a key with  $b(v) \sqsubseteq k$ . For efficient updates, the node  $w$  storing  $k$  has a field  $r(w) = b(v)$ . These  $\text{key}_2$  values allow returning a valid result for a prefix search when stopping at any Patricia node. It is possible to assure that every inner Patricia node with two children has a  $\text{key}_2$  pointing to a leaf node in its subtree.

To allow efficient prefix search, the Patricia Trie has been extended in [14]. Between every pair of directly connected Patricia nodes, Msd nodes (from Most Significant Digit) are added. Their length is chosen in a way that those nodes are hit by a binary search first. More specifically, Msd nodes are inserted between Patricia nodes such that their length is considered first by the binary search before the Patricia nodes around them are considered. We only give a short definition of the calculation of an Msd label in Definition 2. In the special case that an Msd label equals the label of a surrounding Patricia node, no Msd node is needed at that position. For details on how Msd nodes improve the prefix search operation, see [14].

**Definition 2 (Msd Label).** *Let  $a = (a_m, \dots, a_0)$  and  $b = (b_m, \dots, b_0)$  be two binary strings of the same length. Possibly, one of them is filled up with leading zeros to have length  $m+1$ . We define  $\text{msd}(a, b)$  to be the position  $j$  where  $a_j \neq b_j$  and  $a_i = b_i$  for all  $i > j$ . That means,  $\text{msd}(a, b)$  is the most significant bit (digit) at which  $a$  and  $b$  differ.*

*Consider the binary labels  $b(u)$  and  $b(v)$  of two nodes  $u, v$ . Let  $\ell_u = |b(u)|$  and  $\ell_v = |b(v)|$  and without loss of generality let  $\ell_u < \ell_v$ . We define the Msd label  $b(m)$  between  $u$  and  $v$  to be the prefix of  $v$  of length  $\sum_{i=\text{msd}(\ell_u, \ell_v)}^{\lceil \log \ell_v \rceil + 1} (\ell_v)_i \cdot 2^i$ .*

For example, consider  $u, v$  with  $b(u) = 10$  and  $b(v) = 100101$ , where  $\ell_u = |b(u)| = (10)_2$  and  $\ell_v = |b(v)| = (110)_2$ . Then  $\text{msd}(\ell_u, \ell_v) = \text{msd}((010)_2, (110)_2) = 2$ , such that an Msd node  $m$  between  $u$  and  $v$  has label  $b(m) = 1001 \sqsubset b(v)$  of length  $2^2 = 4$ .

The HPT supports operations  $\text{PREFIXSEARCH}(x)$  and  $\text{INSERT}(x)$  for a binary string  $x$  in  $\mathcal{O}(\log |x|)$  read accesses on the hash table. Insertion takes additional  $\mathcal{O}(1)$  write accesses and  $\text{DELETE}(x)$  is supported in constant hash table accesses. Furthermore, the memory space usage is in  $\Theta(\sum_{k \in \text{KEYS}} |k|)$ .

**Modification.** We modify the HPT to simplify the stabilization technique. Consider Fig. 3. The original HPT has a structure as shown on the left side. The Msd node  $m$  is in between the Patricia nodes  $u$  and  $w$  such that  $u$  and  $w$  point to  $m$  and  $m$  points to  $u$  (parent) and  $w$  (child). We modify this structure by having  $u$  and  $w$  point to each other and not to  $m$ . By this, deletions of Msd nodes do not concern the connectivity between Patricia nodes while the advantages of Msd nodes are still present. The crucial property of Msd nodes is that they point to Patricia nodes. Edges towards Msd nodes are not needed for the efficient operations introduced in [14]. For the rest of this paper, when we refer to the HPT, we mean the HPT with this small modification.

Next, we introduce some common terms that are used throughout the paper. HPT is the set of all data nodes of the HPT. This includes PAT as the set of nodes used in the original Patricia Trie and MSD which are the Msd nodes. By definition  $HPT = PAT \cup MSD$ . We denote by KEYS the set of keys stored by the HPT. Let  $u, v \in HPT$  with  $b(u) \sqsubset b(v)$ . In this case we say,  $u$  is *above*  $v$  while  $v$  is *below*  $u$ . Let  $w \in HPT$  such that  $b(u) \sqsubset b(w) \sqsubset b(v)$ . Then  $w$  is in *between*  $u$  and  $v$ . If for two  $u, v \in HPT$  with  $b(u) \sqsubset b(v)$  there is no  $w \in HPT$  with  $b(u) \sqsubset b(w) \sqsubset b(v)$ , then  $u$  and  $v$  are *closest* to each other. We say a child edge  $e$  of  $v \in HPT$  is *valid*, if there exists a node  $w \in HPT$  with  $b(v) \circ e = b(w)$ . Similar, a parent edge  $e$  of  $v \in HPT$  is valid, if there exists a node  $w \in HPT$  with  $b(w) \circ e = b(v)$ . Consider two nodes  $v, u \in HPT$ , where  $u$  has an edge pointing to  $v$  and vice versa. We then speak of a *bidirectional* edge.

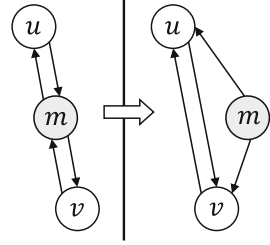


Fig. 3. Modified HPT

### 3 The SHPT Protocol

In the following, we present SHPT, our self-stabilizing protocol for maintaining a HPT. The corrections of SHPT can be divided into several parts. We present our assumptions concerning the underlying DHT first. Afterwards, we give an intuition on the different types of repairs our protocol performs. We often speak about actions executed by a HPT node  $v$ . This translates to actions that are executed by the corresponding DHT node storing  $v$ . For detailed Pseudocode, we refer to [15].

#### 3.1 Properties of the DHT

We assume that the underlying DHT is in a legal state, i.e., it provides the actions DHT-SEARCH( $x$ ) and DHT-INSERT( $x$ ) which are carried out reliably on the stored data. Deletion of data is only done locally by our protocol. Stability of the DHT is crucial as our protocol relies on finding/manipulating nodes of the HPT solely based on their hash value given by their label. There are a lot of different self-stabilizing DHTs presented in the literature. Some of them are mentioned in Sect. 1.2.

Our main demand on the DHT is that at some point nodes are stored such that they can always be retrieved by their labels. HPT nodes are essentially data-items. Every DHT node regularly checks if all its stored data is at the correct peer based on the hashing. If data is stored incorrectly, it is sent towards the correct DHT node. When a data item  $i$  is inserted at a DHT node  $n$ ,  $n$  checks if  $i$  is already present. If yes,  $i$  is only inserted if it does not collide with an already stored Patricia node that stores a key. If a HPT node  $v$  has been inserted, a presentation method is triggered for  $v$  and  $v$  is directly presented to the nodes referred to by  $p_-(v)$ ,  $p_0(v)$  and  $p_1(v)$ . The presentation mechanism is

presented later. This assumption assures that keys are preserved while insertion is not blocked and every HPT node is presented at least once.

### 3.2 Correcting Edge Information

One general problem for self-stabilizing solutions is that every stored information can be corrupted. Thus, our protocol regularly checks information stored in a HPT node. Consider a node  $v \in \text{HPT}$ . We refer to the information provided by the fields  $p_-(v)$ ,  $p_1(v)$  and  $p_0(v)$  as well as  $key_2(v)$  and  $r(v)$  as *edge information*. Edge information can be checked rather simply as it allows reconstruction of a node's label  $b(w)$ . The label can be used to query the DHT for an (incomplete) copy of  $w$ .  $v$  can then compare the information stored at  $w$  with its own and decide for corrections. Some inconsistencies in the local structure can also be checked without querying the DHT. In general, when checking an edge  $e$  at node  $v$ , we distinguish three cases (see Fig. 4):

- (a)  $e$  has a wrong form. For example, if  $p_1(v) = (0\dots)$  or  $p_-(v)$  is not a suffix of  $b(v)$ . In this case, the edge is considered corrupted and is cleared.
- (b) The node  $w$  that  $e$  points to does not exist. Again,  $e$  is not correct and is cleared.
- (c) The node  $w \in \text{HPT}$  that  $e$  points to does exist, but the edge provided by  $w$  which should point to  $v$  does not match  $e$ . Several sub-cases arise here. The protocol may have to simply present  $v$  to  $w$ , or a new node may need to be inserted.

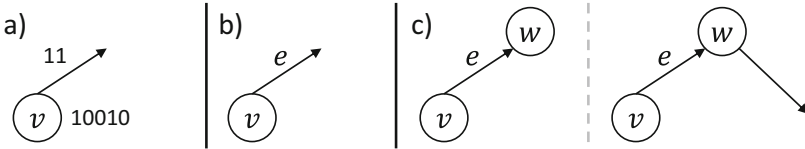


Fig. 4. Examples for the cases of wrong edge information.

Additionally, every node avoids edges pointing to Msd nodes. Such edges are treated as if they pointed to a non-existing node. A node  $v$  can check the values of  $p_-(v)$ ,  $key_2(v)$  and  $r(v)$  by calculating if the prefix relation between itself and the respective nodes fulfills the definition of the hashed Patricia Trie. To prevent the spreading of incorrect information, new edges are only stored if they comply with the definition of the hashed Patricia Trie from the local perspective of  $v$ . We will go into detail on the creation of new edges and the insertion of nodes later.

### 3.3 Maintaining Connections

Our goal to stabilize the Patricia nodes of a HPT can also be formulated using *Branch Sets* as described in Definition 3. A Branch Set consists of all Patricia nodes on a branch from the root to a leaf node (see Fig. 5). When the HPT is in a legal state, there are as many Branch Sets as there are leaf nodes.

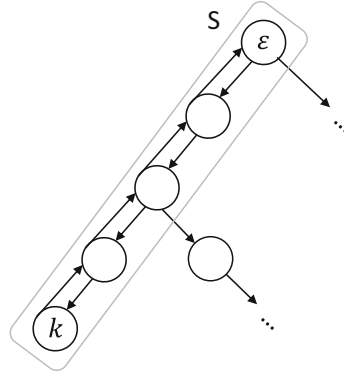
**Definition 3 (Branch Set).** Consider a set of Patricia nodes with maximum cardinality  $S$  such that  $u, w \in S$  implies  $b(u) \sqsubset b(w)$  or  $b(w) \sqsubset b(u)$  and the Patricia node  $v \in S$  with maximum label length stores a key  $k$ . We call this set the Branch Set of  $k$ .

We apply a technique called Linearization [17] to all Patricia nodes to create a list sorted by label length for all Branch Sets in finite time. It is important to exclude Msd nodes from the Linearization. Msd nodes are not presented nor do they delegate presentation messages. Due to deletion of a Patricia node, an Msd node might still be presented accidentally. However, we limit this problem by carefully handling deletions and insertions as described later. For the Linearization to work, we need to make sure that all nodes in a Branch Set are brought into and kept in a weakly connected state.

A Patricia node  $v$  with an empty parent edge tries to recreate connectivity by doing a modified PREFIXSEARCH( $b(v)$ ) similar to the one presented in [14]. The procedure we call BINARYPREFIXSEARCH( $b(v)$ ) does not search for  $b(v)$  itself and only consists of the binary search phase of the PrefixSearch( $x$ ) of [14], returning a copy of a Patricia node  $w$  with  $b(w) \sqsubset b(v)$ . If no such node exists, we conclude that the root node is non-existent and trigger a construction of it.

Further, we let every Patricia node present its own label to its parent and its two children using a *presentation message*. A message presenting  $v$  is delegated to the Patricia node  $w$  closest to  $v$ . Delegation happens only by using edges and intermediate nodes sharing a Branch Set with  $v$ . All nodes maintain connections to labels which are closest to them while delegating presentations of other labels. This behavior resembles the Linearization approach presented in [17], allowing our protocol to form a sorted list for all branches of the HPT.

There is still an important issue we need to resolve. Consider a Branch Set  $S$  of nodes. We can end up in situations where nodes exist that do not contribute to the hashed Patricia Trie. Such nodes can be Patricia nodes not storing a key. To reduce memory demands, we are interested in removing unneeded nodes. In principle, deletion without harming connectivity can be done since the root node is always known implicitly. However, deletion increases distances. In addition,



**Fig. 5.** Branch Set  $S$  from the root ( $\epsilon$ ) to a leaf node ( $k$ ) is the set of nodes in a branch of the hashed Patricia Trie in a legal state.



our protocol must provide the ability to create and integrate new Patricia nodes. When inserting and deleting nodes, we need to make sure that no loops are possible in which the system may take forever to stabilize. We will explain how to avoid such loops in the following.

### 3.4 Removal/Creation of Nodes

Due to the implicitly known root node, deletion is possible and should be considered to reduce memory demands. We distinguish between Msd nodes and Patricia nodes. Our modification allows us to handle Msd nodes in a simple and efficient way. We try to avoid any edges pointing to Msd nodes such that eventually, deletion and creation of Msd nodes does not influence the Patricia nodes and their structure. Only if there are two Patricia nodes  $u, w$  connected via a bidirectional edge, an Msd node between them might be inserted. Fortunately, Msd labels can be calculated locally and a corresponding Msd node can easily be accessed by querying the DHT. Any Msd node which is not between such two Patricia nodes, or has an incorrect label, is deleted.

A Patricia node  $v$  (except for the root) is *unnecessary* if  $key(v) = nil$  and there are no two Patricia nodes  $u, w$ , both storing a key, such that  $b(v) = lcp(b(u), b(w))$ , i.e.,  $u$  should be in a different subtree than  $w$  below  $v$ . From a global point of view, we can easily decide if  $v$  is unnecessary solely based on information about the situation below  $v$ . From a local perspective,  $v$  cannot decide but only assume to be unnecessary if it lacks child edges. We make the local protocol aggressive by deleting any node that lacks child edges and assumes to be unnecessary. This also introduces deletion of necessary Patricia nodes. Therefore, we always trigger a creation of new HPT nodes by Patricia nodes below the new ones. This avoids loops of creation and deletion of nodes, because newly created nodes inherently have valid children and, thus, do not assume to be unnecessary. Patricia nodes storing a key essentially form a stable starting point, because they are never deleted. The need to insert a Patricia node is detected by comparing a node's parent edge with the corresponding edge provided by the parent.

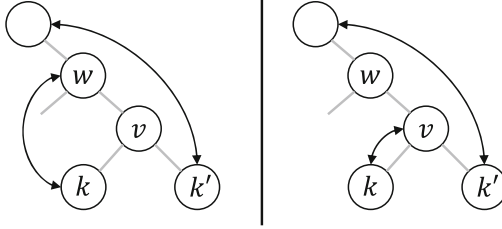
### 3.5 Distribution of References to Keys

In addition, SHPT tries to achieve the following. Every inner Patricia node  $v$  with two children should store a  $key_2(v) = b(w)$  which points to a leaf node  $w$  storing a key such that  $b(v) \sqsubset b(w)$ . The respective leaf node  $w$  stores an  $r(w)$  value pointing to  $v$ . This property is helpful for efficient prefix search. No matter at which Patricia node the prefix search stops, there is a key referenced having the node's label as a prefix. This key is a valid result for the search query. We call all inner Patricia nodes with two children and the root node *key<sub>2</sub> nodes*. Due to the resemblance of the hashed Patricia Trie with a binary tree, Fact 1 holds.

**Fact 1.** *Let  $L$  be the number of leaf nodes. Let  $I$  be the number of key<sub>2</sub> nodes. When the HPT is in a legal state, it holds  $I \leq L \leq I + 1$ .  $L = I$ , if the root has one child and  $L = I + 1$  if it has two.*

To assure that every leaf node is referenced by a  $\text{key}_2$  node, we allow the root to store up to two  $\text{key}_2$  values. This reduces the number of hash table accesses created by our protocol, when the HPT is in a legal state.

If we naively assign leaf nodes to  $\text{key}_2$  nodes, this may lead to situations in which a  $\text{key}_2$  node cannot get a  $\text{key}_2$  value. For an example, consider Fig. 6. The critical observation is that  $\text{key}_2$  nodes with a shorter label, in general, have more possible leaf nodes they can point to than  $\text{key}_2$  nodes with a longer label. Therefore, our protocol aims at prioritizing  $\text{key}_2$  nodes which are closer to leaf nodes.



**Fig. 6.** Example where  $v$  cannot get a  $\text{key}_2$  (left). The leaf nodes  $k$  and  $k'$  storing a key are already associated to Patricia nodes above  $v$ . The blocking of  $v$  is resolved as  $v$  takes over the  $\text{key}_2$  of  $w$  (right).

We divide the protocol into three parts. First, all nodes continuously check if they should store a  $\text{key}_2$  or  $r$  value and whether such a value points to a leaf node, respectively  $\text{key}_2$  node. Second, if a leaf node  $v$  does not store a value in  $r(v)$ , it presents its label upwards in the HPT by sending a message crossing only parent edges. The first  $\text{key}_2$  node  $w$  without a  $\text{key}_2$  receiving the message sets  $\text{key}_2(w) = b(v)$ . Third, a  $\text{key}_2$  node  $v$  repairs in the following way. If  $\text{key}_2(v)$  points to leaf node  $w$  with  $b(v) \sqsubset b(w)$ , there are two cases.

- (a)  $b(v) \sqsubset r(w)$ : Then  $\text{key}_2(v)$  is set to  $\text{nil}$  since there may already be some  $\text{key}_2$  node with longer label pointing at  $w$ .
- (b) Else,  $v$  has either longer label than  $r(w)$  or  $r(w) = \text{nil}$ . The protocol sets  $r(w) = b(v)$ .

If  $\text{key}_2(v) = \text{nil}$ , a message is sent upwards in the HPT and the first  $\text{key}_2$  node  $w$  with  $b(v) \sqsubset \text{key}_2(w)$  responds to  $v$ . Then,  $\text{key}_2(v)$  is set to  $\text{key}_2(w)$ . Eventually,  $v$  takes over the  $\text{key}_2$  value of  $w$ , because  $w$  executes case (a).

Intuitively,  $\text{key}_2$  nodes without a  $\text{key}_2$  pull values from nodes with shorter label. Simultaneously, leaf nodes without an  $r$  value present their label towards the root.

## 4 Protocol Analysis

In this section, we show that SHPT is self-stabilizing and transforms the HPT in finite time to a legal state. Furthermore, we present results concerning memory

usage and the number of hash table accesses and messages when the HPT is in a legal state.

#### 4.1 Correctness

We begin by showing the correctness of our self-stabilizing protocol. We use a commonly known technique introduced by Dijkstra in [7]. Our goal is to show Theorem 1. For that we consider a sequence of intermediate states that are reached consecutively until the HPT is in a legal state. For every state we show *convergence* towards the state and *closure* within it, i.e., the properties of the state are kept by our protocol.

**Theorem 1.** *The algorithm creates in finite time a hashed Patricia Trie in a legal state out of any initial state in which the DHT is in a legal state and there is a set of unique keys stored at DHT nodes.*

In the following, we briefly sketch the main proof by presenting a sequence of main lemmas that roughly reflect the states the system reaches. Each main lemma thereby consists of multiple properties that are proven by a set of lemmas on its own. The full proof consisting of all lemmas, their respective proofs, and the complete definition of a legal state of the HPT can be found in [15].

To prove the correctness captured in Theorem 1, we first need to formally define a legal state of the HPT. Due to space limitations, we only give an intuitive definition. For the complete definition, see the full version [15]. Intuitively, the HPT is in a legal state if we have as few HPT nodes as possible in the system, all keys are stored correctly, the structure is consistent to the (modified) definition presented in Sect. 2, and the references to keys in  $\text{key}_2$  nodes are existing and stored at correct nodes.

Initially, we only assume that a set of unique keys is stored at DHT nodes. The first lemma states that general repair mechanisms assure correctly stored keys and Patricia nodes.

**Lemma 1.** *In finite time it holds: Every key  $k$  is stored in a node  $v \in \text{PAT}$  with  $b(v) = k$ . Furthermore, every node is stored at the DHT node responsible for it. Consider any  $v \in \text{HPT}$  that is deleted. As long as  $v$  is not reconstructed, in finite time it holds:*

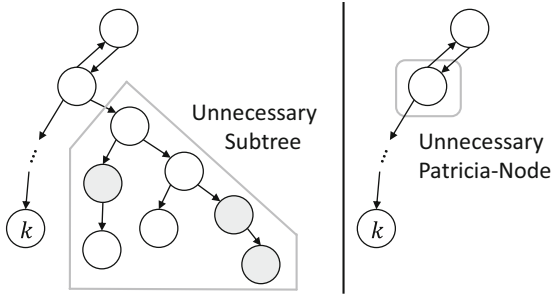
- (a) *There is no presentation message for  $b(v)$ .*
- (b) *There is no edge pointing towards  $b(v)$  in the system.*

From now on, the proof consists of three phases. In a first phase, all Patricia nodes which are not needed for the final structure are removed. The second phase considers the reconstruction of the binary tree structure of the HPT and corrects the sets of Patricia nodes and Msd nodes. In the third and last phase, information stored in  $\text{key}_2$  and  $r$  fields is made consistent.

### Phase I – Deletion of Patricia Nodes

In this phase, the protocol makes sure that all Patricia nodes which are not needed in the final structure are removed. Initially, information stored at HPT nodes that directly contradicts the definition of the HPT is cleared. This can be information such as a parent edge at  $v \in \text{HPT}$  that is no suffix of  $b(v)$ . After that, Patricia nodes and Msd nodes in unnecessary subtrees, i.e., subtrees not containing a key, and unnecessary inner Patricia nodes are gradually removed (Fig. 7). Every leaf node in an unnecessary subtree detects in finite time that it has no valid children and is deleted.

**Lemma 2.** *In finite time, every unnecessary Patricia node is removed. A Patricia node  $v$  is unnecessary if there are no two keys  $k_1$  and  $k_2$  with  $b(v) = \text{lcp}(k_1, k_2)$ .*



**Fig. 7.** Node  $k$  stores a key. Msd nodes are sketched in grey. First, unnecessary subtrees are deleted (left), then remaining unnecessary Patricia nodes are removed (right).

Patricia nodes which are necessary may still be deleted because of their local perspective. However, this deletion is limited and stops after finitely many deletions. This holds, because Patricia nodes are only deleted due to incorrect child edges. If a new Patricia node with a long label is inserted, its child edges are initially valid and stay valid. There cannot be infinitely many deletions triggered, because the structure stabilizes bottom-up.

**Lemma 3.** *In finite time, every Patricia node has valid child edges pointing to Patricia nodes and no further Patricia node is deleted.*

### Phase II – Reconstruction

In the second phase, SHPT reconstructs the HPT by rebuilding missing Patricia nodes and repairing connections. Since every node tries to create a parent edge pointing to a Patricia node with shorter label, eventually all missing Patricia nodes are detected and can be inserted. The process works in a bottom-up fashion, i.e., Patricia nodes with longer labels reconstruct missing nodes with shorter ones. The Patricia nodes storing a key as well as the root node act as fixed points in this case, because they are never deleted once constructed.

**Lemma 4.** *In finite time, the root node exists and no Patricia node points to an Msd node. Furthermore, missing Patricia nodes are reconstructed. Also, every Patricia node has valid edges pointing only to existing Patricia nodes, i.e., there is a path from every Patricia node to the root and there is a path from the root to every Patricia node.*

It is crucial that no Patricia node points to an Msd node, because edges to Msd nodes are effectively treated as corrupt ones. This property assures that Msd nodes are eventually excluded from the Linearization procedure. Linearization then allows us to show that every Branch Set (see Definition 3) of Patricia nodes eventually forms a stable sorted list. Incorrect Msd nodes are removed without affecting the rest of the HPT and missing Msd nodes are inserted. Further, correct Msd nodes are not deleted, because the two Patricia nodes closest to a correct Msd node are not deleted and do not change their edges any more. All these properties are reflected in Lemma 5. For completeness, we refer to the definition of incorrect and missing Msd nodes in the full proof in [15].

**Lemma 5.** *In finite time for every Branch Set  $S$  it holds: Between every pair of closest Patricia nodes  $u, w \in S$  there is a bidirectional edge. Furthermore, every incorrect Msd node is removed and all missing Msd nodes are inserted.*

### Phase III – Consistency

In the final phase the information stored in  $\text{key}_2$  and  $r$  fields is corrected to be consistent. Due to Fact 1, we know that this can be achieved. The root is allowed to store up to two  $\text{key}_2$  values. Therefore, there is always a way to store all keys of leaf nodes in  $\text{key}_2$  nodes. First, we show that nodes which should not store a  $\text{key}_2$  value remove any such stored value. Further, references in  $\text{key}_2$  and  $r$  fields are deleted when they contradict the relationship  $r(\text{key}_2(v)) = b(v)$ , where  $v$  is a  $\text{key}_2$  nodes and  $\text{key}_2(v)$  references a leaf node.

**Lemma 6.** *In finite time, only  $\text{key}_2$  nodes store a  $\text{key}_2$  and only leaf nodes store an  $r$  value. Every  $\text{key}_2$  value stored at a Patricia node  $v$  points to a leaf  $w$  with  $b(v) \sqsubset b(w)$  and every  $r$  value stored at a Patricia node  $w$  points to a  $\text{key}_2$  node  $v$  with  $b(v) \sqsubset b(w)$ .*

From now on,  $\text{key}_2$  nodes not storing a  $\text{key}_2$  try to acquire the  $\text{key}_2$  of a  $\text{key}_2$  node above them. Leaf nodes lacking a reference in  $r$  present themselves to  $\text{key}_2$  nodes above them. Therefore, the length of the longest label of a  $\text{key}_2$  node not storing a staying  $\text{key}_2$  reduces over time. As this length is finite, the process terminates. Thereafter, the  $r$  values of leaf nodes are corrected, because the  $\text{key}_2$  values do not change any more.

**Lemma 7.** *In finite time, all  $\text{key}_2$  nodes store a stable  $\text{key}_2$  and all leaf nodes store a stable  $r$  value. For every  $\text{key}_2$  node  $v$ , the node  $w$  with  $b(w) = \text{key}_2(v)$  is a leaf node with  $r(w) = b(v)$ .*

Finally, our protocol is correct as all unnecessary nodes are removed, missing nodes are inserted, Patricia nodes are connected by bidirectional edges, and the information stored in  $\text{key}_2$  and  $r$  fields is consistent such that the HPT is in a legal state in finite time.

## 4.2 Overhead

Assume, the HPT is in a legal state. We give results for the complexity in terms of hash table accesses and messages and the memory overhead of our solution. Due to space limitations, we refer to the full version [15] for the proofs of the following theorems. When a DHT node executes SHPT by calling its TIMEOUT Method, exactly one HPT node is checked. Thereby, at most a constant number of other HPT nodes may be partially acquired or notified and Theorem 2 holds.

**Theorem 2.** *When the HPT is in a legal state, SHPT creates a constant number of hash table (read) accesses and messages per call of TIMEOUT at each DHT node.*

Unnecessary Patricia nodes and incorrect Msd nodes are removed by SHPT. Therefore, the HPT nodes are the same as presented in the construction in Sect. 2 and Theorem 3 holds.

**Theorem 3.** *Let  $d$  be the number of bits needed to store all keys. The total memory used by a HPT in a legal state is in  $\Theta(d)$  bits.*

## References

1. Afek, Y., Kuten, S., Yung, M.: Memory-efficient self stabilizing protocols for general networks. In: van Leeuwen, J., Santoro, N. (eds.) WDAG 1990. LNCS, vol. 486, pp. 15–28. Springer, Heidelberg (1991). [https://doi.org/10.1007/3-540-54099-7\\_2](https://doi.org/10.1007/3-540-54099-7_2)
2. Arora, A., Gouda, M.: Distributed reset. *IEEE Trans. Comput.* **43**(9), 1026–1038 (1994)
3. Awerbuch, B., Varghese, G.: Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In: Proceedings 32nd Annual Symposium of Foundations of Computer Science, pp. 258–267. IEEE, October 1991
4. Clouser, T., Nesterenko, M., Scheideler, C.: Tiara: a self-stabilizing deterministic skip list and skip graph. *Theor. Comput. Sci.* **428**, 18–35 (2012)
5. Collin, Z., Dolev, S.: Self-stabilizing depth-first search. *Inf. Process. Lett.* **49**(6), 297–301 (1994)
6. Cramer, C., Fuhrmann, T.: Self-stabilizing ring networks on connected graphs. Technical report, University of Karlsruhe (2005)
7. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. *Commun. ACM* **17**(11), 643–644 (1974)
8. Dolev, S., Kat, R.I.: HyperTree for self-stabilizing peer-to-peer systems. In: 3rd IEEE International Symposium on Proceedings of the Network Computing and Applications. NCA 2004, pp. 25–32. IEEE Computer Society, Washington, DC (2004)
9. Flatebo, M., Datta, A.K.: Two-state self-stabilizing algorithms for token rings. *IEEE Trans. Softw. Eng.* **20**(6), 500–504 (1994)
10. Gonnet, G.H., Baeza-Yates, R.A., Snider, T.: New indices for text: PAT Trees and PAT arrays. In: Frakes, W.B., Baeza-Yates, R. (eds.) *Information Retrieval*, pp. 66–82. Prentice-Hall Inc., Upper Saddle River (1992)

11. Jacob, R., Richa, A., Scheideler, C., Schmid, S., Täubig, H.: SKIP+: a self-stabilizing skip graph. *J. ACM* **61**(6), 36 (2014)
12. Jacob, R., Ritscher, S., Scheideler, C., Schmid, S.: A self-stabilizing and local Delaunay graph construction. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 771–780. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-10631-6\\_78](https://doi.org/10.1007/978-3-642-10631-6_78)
13. Kniesburges, S., Koutsopoulos, A., Scheideler, C.: Re-chord: a self-stabilizing chord overlay network. In: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures. SPAA 2011, pp. 235–244. ACM, New York (2011)
14. Kniesburges, S., Scheideler, C.: Hashed Patricia Trie: efficient longest prefix matching in peer-to-peer systems. In: Katoh, N., Kumar, A. (eds.) WALCOM 2011. LNCS, vol. 6552, pp. 170–181. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-19094-0\\_18](https://doi.org/10.1007/978-3-642-19094-0_18)
15. Knollmann, T., Scheideler, C.: A Self-stabilizing hashed Patricia Trie. ArXiv e-prints. <http://arxiv.org/abs/1809.04923> (2018)
16. Morrison, D.R.: PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *J. ACM* **15**(4), 514–534 (1968)
17. Onus, M., Richa, A., Scheideler, C.: Linearization: locally self-stabilizing sorting in graphs. In: Proceedings of the Meeting on Algorithm Engineering & Experiments, pp. 99–108. Society for Industrial and Applied Mathematics, Philadelphia (2007)
18. Rowstron, A., Druschel, P.: Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 329–350. Springer, Heidelberg (2001). [https://doi.org/10.1007/3-540-45518-3\\_18](https://doi.org/10.1007/3-540-45518-3_18)
19. Shaker, A., Reeves, D.S.: Self-stabilizing structured ring topology P2P systems. In: Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing, pp. 39–46. IEEE, August 2005
20. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.* **11**(1), 17–32 (2003)