Distrib. Comput. (2008) 21:141–161
DOI 10.1007/s00446-008-0061-5

Compact separator decompositions in dynamic trees and applications to labeling schemes

Amos Korman · David Peleg

Received: 1 October 2007 / Accepted: 4 March 2008 / Published online: 1 April 2008

Abstract This paper presents an efficient scheme maintaining a *separator decomposition representation* in dynamic trees using asymptotically optimal labels. In order to maintain the short labels, the scheme uses relatively low message complexity. In particular, if the initial dynamic tree contains only the root, then the scheme incurs an $O(\log^4 n)$ amortized message complexity per topology change, where n is the current number of vertices in the tree. As a separator decomposition is a fundamental decomposition of trees used extensively as a component in many static graph algorithms, our dynamic scheme for separator decomposition may be used for constructing dynamic versions to these algorithms. The paper then shows how to use our dynamic separator decomposition to construct efficient labeling schemes on dynamic trees, using the same message complexity as our dynamic separator scheme. Specifically, we construct efficient routing schemes on dynamic trees, for both the designer and the adversary port models, which maintain optimal labels, up to a multiplicative factor of $O(\log \log n)$. In addition, it is shown how to use our dynamic separator decomposition scheme to construct dynamic labeling schemes supporting the ancestry and NCA relations using asymptotically optimal labels, as well as to extend a known result on dynamic distance labeling schemes.

Supported in part at the Technion by an Aly Kaufman fellowship.
Supported in part by a grant from the Israel Science Foundation.

A. Korman (✉)
Information Systems Group, Faculty of IE&M,
The Technion, Haifa 32000, Israel
e-mail: pandit@tx.technion.ac.il

D. Peleg
Department of Computer Science and Applied Mathematics,
The Weizmann Institute of Science, Rehovot 76100, Israel
e-mail: david.peleg@weizmann.ac.il

Keywords Distributed algorithms · Dynamic networks · Routing schemes · Graph decompositions · Informative labeling schemes

1 Introduction

Background : A distributed representation scheme is a scheme maintaining global information using local data structures (or *labels*). Such schemes play an extensive and sometimes crucial role in the fields of distributed computing and communication networks. Their goal is to locally store useful information about the network and make it readily and conveniently accessible. As a notable example, the basic function of a communication network, namely, message delivery, is performed by its *routing scheme*, which in turn requires maintaining certain topological knowledge. Often, the performance of the network as a whole may be dominated by the quality of the routing scheme and the accuracy of the topological information. Representation schemes in the *static* (fixed topology) setting were the subject of extensive research, cf. [1,5,6,9,13,15]. The common measure for evaluating a static representation scheme is the *label size*, i.e., the maximum number of bits used in a label. In this paper, a representation scheme with asymptotically optimal label size is termed *compact*.

The more realistic (and more involved) distributed *dynamic* setting, where processors may join or leave the network or new connections may be established or removed, has received much less attention. Clearly, changes in the network topology may necessitate corresponding changes in the representation. Consequently, in the distributed dynamic setting, an *update protocol* is activated where the topology change occurs, and its goal is to update the vertices, by transmitting messages over the links of the underlying network.

Ideally, the update protocol maintains short labels using only a limited number of messages.

In this paper we consider representation schemes in dynamic trees, operating under the *leaf-dynamic* tree model, in which at each step, a leaf may either join or leave the tree. We consider the controlled dynamic model, which was also considered in [3,16], in which the topological changes do not occur spontaneously. Instead, when an entity wishes to implement a topology change at some vertex u, it enters a *request* at u, and performs the change only after the request is granted a permit from the update protocol. See Sect. 2 for a formal definition. The controlled model may be found useful in Peer to Peer applications and in other popular overlay networks. See [16] for more details and motivations regarding the controlled model.

We present several dynamic representation schemes, which are efficient in both their label size and their communication complexity. In particular, if the initial tree contains only the root, then all our dynamic schemes incur $O(\log^4 n)$ amortized message complexity, per topological change. We first present a compact representation scheme of a separator decomposition in dynamic trees, and then use this basic compact scheme to derive compact labeling schemes supporting the ancestry and NCA relations on dynamic trees. In addition, we present dynamic routing schemes which have optimal label size up to $O(\log \log n)$ multiplicative factor, for both the adversary and the designer port models. Finally, we show how to use our dynamic separator decomposition to extend a known result on dynamic distance labeling schemes.

Related work: An elegant and simple compact labeling scheme was presented in [13], for supporting the ancestry relation on static n-vertex trees using labels of size $2 \log n$. Applications to XML search engines motivated various attempts to improve the constant multiplicative factor in the label size, see [1,2].

Static compact labeling schemes were presented for two types of NCA relations on trees. For the id-based NCA relation (which is the type of NCA relation we consider in this paper), a static labeling scheme was developed in [21] using labels of $\Theta(\log^2 n)$ bits on n-vertex trees. A static labeling scheme supporting the label-based NCA relation using labels of $\Theta(\log n)$ bits on n-vertex trees was presented in [5]. In addition, [5] gave a survey on applications and previous results concerning NCA queries on trees, in both the distributed and centralized settings.

Labeling schemes for routing on static trees were investigated in a number of papers until finally optimized in [9,10,26]. For the *designer port* model, in which the designer of the scheme can freely enumerate the port numbers of the vertices, [9] shows how to construct a routing scheme using labels of size $O(\log n)$ on n-vertex trees. In the *adversary port* model, where the port numbers are fixed by an adversary, it is shown how to construct a routing scheme using labels of size $O(\log^2 n / \log \log n)$ on n-vertex trees. In [10] it is shown that both label sizes are asymptotically optimal. Independently, a routing scheme for trees of label size $(1+o(1)) \log n$ was given in [26] for the designer port model.

Dynamic data structures for trees have been studied extensively in the centralized model, cf. [7,23,24]. For comprehensive surveys on centralized dynamic graph algorithms see [11,22].

A survey on popular link state dynamic routing protocols (e.g., OSPF) can be found in [25]. Compared to our dynamic routing schemes, these routing schemes are more robust on weaker dynamic models, such as ones which allow spontaneous faults; however, their message complexity is higher.

The controlled model is presented in [3], which also establishes an efficient dynamic controller that can operate in the *leaf-increasing* tree model, where the only topology change allowed is that of a leaf joining the tree. This controller can, in particular, be used to maintain a constant approximation of the number of vertices in the (leaf-increasing) tree, using $O(n \log^2 n)$ messages, where n is the final (and maximum) number of vertices. In [16] an extended controller was derived for the controlled model, which can operate under both insertions and deletions of both leaves and internal vertices. In particular, that controller can be used to efficiently maintain a constant approximation of the number of vertices in the dynamic tree, undergoing both deletions and additions of vertices, using low message complexity. Specifically, the approximation scheme incurs $O(n_0 \log^2 n_0) + O\left(\sum_j \log^2 n_j\right)$ messages, where n_0 is the initial tree size, and n_j is the size of the tree immediately after the j'th topology change. Note that if the initial tree contains just the root, then this complexity can be considered as $O(\log^2 n)$ amortized message complexity per topology change. Moreover, if the tree can only grow, then the message complexities of the schemes in [3,16], are the same.

A dynamic routing scheme in the leaf-increasing tree model was given in [4] using identities of size $O(\log^2 n)$, database size $O(\Delta \log^3 n)$ (where Δ is the maximum degree in the tree) and message complexity $O(n \log n)$, where n is the final number of vertices. In our terminology, the scheme of [4] uses label size $O(\Delta \log^3 n)$ and message complexity $O(n \log n)$.

Dynamic distance labeling schemes on trees were presented in [17,19] for the *serialized model*, in which it is assumed that the topology changes are spaced enough so that the update protocol can complete its operation before the next topology change occurs (the serialized model is a more restricted model than the controlled model, considered in this paper). A compact distance labeling scheme with amortized message complexity $O(\log^2 n)$ for unweighed dynamic trees operating under the leaf-dynamic tree model was presented in [19]. For the case where the tree is weighted, two

dynamic β-approximate distance labeling schemes (in which given two labels, one can infer a β-approximation to the distance between the corresponding vertices) were presented in [17]. The first scheme applies to a model in which the tree topology is fixed but the edge weights may change, and the second applies to a model in which the only topological event that may occur is that of an edge increasing its weight by one. The amortized message complexity of the first scheme depends on the local density parameter of the underlying tree and the amortized message complexity of the second scheme is polylogarithmic. Both schemes have label size $O(\log^2 n + \log n \log W)$ where W denotes the largest edge weight in the tree.

Two general translation methods for extending static labeling schemes on trees to the dynamic setting were considered in [14,19], also for the (rather restrictive) serialized model. Both approaches fit a number of natural functions on trees, such as ancestry relation, routing, NCA relation etc. The translation methods incur some overheads over the static scheme, in both the label size and the message complexity. Specifically, the method of [14] yields dynamic compact labeling schemes, although the amortized message complexity is high, namely, $O(n^\epsilon)$. On the other hand, the label sizes of the dynamic labeling schemes in [14], which use polylogarithmic amortized message complexity, have a multiplicative overhead factor of $O(\log n / \log\log n)$ over the optimal size.

Our contributions: In this paper we consider a dynamic tree T operating under the leaf-dynamic tree model and the controlled model, and present several efficient dynamic schemes for T. All our schemes incur $O(n_0 \log^4 n_0) + O\left(\sum_j \log^4 n_j\right)$ messages, where n_0 is the initial number of vertices and n_j is the number of vertices immediately after the j'th topology change. Note that if the initial tree contains only the root (as assumed in [14]), then the amortized message complexity is $O(\log^4 n)$ per topological change, where n is the current number of vertices in T.

We first present an efficient protocol maintaining a compact separator decomposition representation in T. Let us note that the general translation method of [14] may also yield such a dynamic compact scheme, however, the resulting scheme uses high amortized message complexity, namely, $O(n^\epsilon)$.

Our basic dynamic separator scheme is then used in order to construct several other dynamic labeling schemes for the dynamic tree T, which improve known results. Specifically, we present dynamic compact labeling schemes supporting the ancestry and the NCA relations, and establish routing schemes for both the designer and the adversary port models, which use optimal label size up to a multiplicative $O(\log \log n)$ factor. For any of the above mentioned functions f, the best known label size for dynamic labeling schemes supporting f, that use polylogarithmic amortized message complexity, has a multiplicative overhead of $O(\log n / \log \log n)$ over the optimal label size. In addition, the best known amortized message complexity for dynamic compact labeling schemes supporting f is $O(n^\epsilon)$.

Finally, we show that our dynamic separator decomposition can also be used to allow the dynamic distance labeling schemes of [17] to operate under a more general dynamic model. In addition to allowing the edges of the underlying tree to change their weight, the extended dynamic model allows also leaves to be added to or removed from the tree. The extended scheme incurs an extra $O\left(n_0 \log^4 n_0\right) + O\left(\sum_j \log^4 n_j\right)$ additive factor to the message complexity.

Overview: Our compact separator scheme is based on an adaptation of our static scheme (described in Sect. 3) which also uses $\Theta(\log n)$ bit labels. The adaptation requires maintaining estimates on the sizes of the various subtrees managed in the decomposition, manipulating and reorganizing these subtrees, and maintaining the corresponding labels and topological data. Generally speaking, each separator v maintains a constant approximation to the number of vertices in the subtree $T^*(v)$ for which v was chosen as a separator. Whenever, the size of $T^*(v)$ grows by some constant factor, the main Protocol DYN_SEP invokes Protocol SHUFFLE on $T^*(v)$, which calculates a new separator decomposition on $T^*(v)$ that is consistent with the global separator decomposition. The correct operation of these protocols relies on the assumption that certain properties hold at the beginning of their execution, and in turn, each of these components guarantees that certain properties hold upon their termination. Hence the correctness proof of the entire algorithm depends on establishing an intricate set of invariants and showing that these invariants hold throughout the execution.

Our dynamic ancestry and routing labeling schemes are based on simple modifications of our dynamic separator scheme. Specifically, in the ancestry scheme, we modify the label of each vertex v by adding two bits per separator s of v, indicating whether s is a descendant of v, ancestor of v or neither. Then, given the labels of two vertices u and v, we first find the last separator s_{last} common to both u and v, using the separator scheme. This separator s_{last} must be on the path connecting u and v and therefore, we can now determine whether u is an ancestor of v, according to whether u is an ancestor of s_{last} and whether v is a descendant of s_{last}. Moreover, since the labels of the separator scheme are only slightly modified, the label size remains $O(\log n)$, which is asymptotically optimal. It turns out that maintaining these slightly modified labels requires only a minor change in the update protocol of our dynamic separator scheme, and does not effect its asymptotic message complexity.

In our routing schemes, we modify the label of each vertex v by also encoding, for each separator s of v, the port number

at v leading from v to the next vertex on the (shortest) path connecting v and s and the port number at s leading from s to the next vertex on the path connecting s and v. Then, given the labels of two vertices u and v, we first find the last separator s_{last} common to both, using the separator scheme. If this separator s_{last} is u, then the port number at u leading to the next vertex on the path from u to v, is found in v's label. Otherwise, if $s_{\text{last}} \neq u$, then the port number at u leading to the next vertex on the path from u to v, is also the port number at u leading to the next vertex on the path from u to s_{last}, which is found in u's label. In the case where the port numbers are given by an adversary, each label uses $O(\log n)$ bits per separator, which sums to $O(\log^2 n)$ bits per label. However, in the designer port model, the sizes of the labels can be reduced to $O(\log n \cdot \log\log n)$. This gives a multiplicative factor of $O(\log\log n)$ over the optimal label size in both port models. As in the dynamic ancestry labeling scheme, maintaining the slightly modified labels can be done using the same asymptotic message complexity as our dynamic separator scheme.

Finally, our dynamic NCA labeling scheme is based on an adaptation of the static scheme of [21], using our dynamic ancestry labeling scheme instead of the static ancestry scheme used therein, and employing the dynamic heavy-child decomposition from [19]. Our extended dynamic distance labeling schemes are rather straightforward combinations of our dynamic separator scheme and the distance labeling schemes in [17].

2 Preliminaries

Our communication network model is restricted to tree topologies. Let T be a tree rooted at vertex r and let $T(v)$ denote the subtree of T rooted at v. For every vertex $v \in T$, let $D(v)$ denote the *depth* of v, i.e., 1 plus the unweighed distance between v and the root r (in particular, $D(r) = 1$). For a non-root vertex v, denote by $p(v)$ its parent in the tree. The *ancestry* relation is defined as the transitive closure of the parenthood relation. Define the *weight* of the vertex v, denoted $\omega(v)$, as the number of vertices in $T(v)$, i.e., $\omega(v) = |T(v)|$. Let n denote the number of vertices in the tree, i.e., $n = \omega(r)$. The ports at each vertex (leading to its different neighbors) are assigned unique *port-numbers*. The enumeration of the ports at a vertex v is known only to v.

For every two numbers $a < b$, let $[a, b)$ denote the set of integers $a \leq i < b$. For every integer $q \geq -1$ let $I_q = [2^{q+3}, 2^{q+4})$ and for every $m \leq n$ and $-1 \leq q \leq \log m$, let $J_q(m) = [\frac{m}{2^{q+1}}, \frac{m}{2^q})$ and let $\widehat{J}_q(m) = [\frac{m}{2^{q+2}}, \frac{m}{2^{q-1}})$. In other words, $\widehat{J}_q(m) = J_{q+1}(m) \cup J_q(m) \cup J_{q-1}(m)$.

Separator decomposition: We first define a *separator decomposition* of a tree T recursively as follows. At the first stage we choose some vertex v in T to be the *level-1* separator of T. Removing v breaks T into disconnected subtrees which are referred to as the subtrees *formed* by v. Each such subtree is decomposed recursively by choosing some vertex to be a level-2 separator, etc.

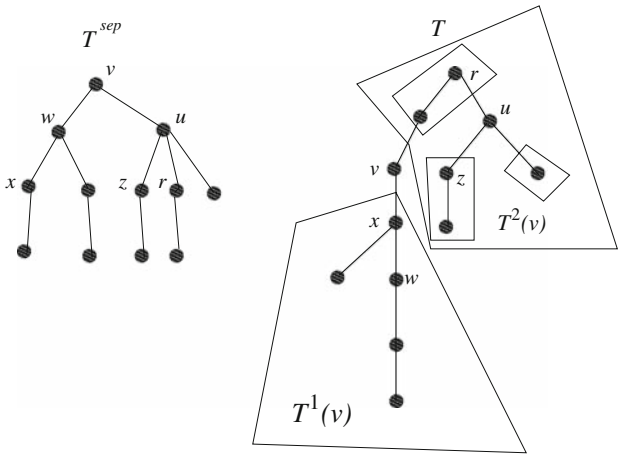

Fig. 1 In the depicted tree T, rooted at r, the vertex v is the level-1 separator of T. Deleting v breaks T into $T^1(v)$ and $T^2(v)$. Similarly, w is the separator of $T^1(v)$ and u is the separator of $T^2(v)$, therefore w and u are the children of v in T^{sep}. Deleting u breaks $T^2(v)$ into three subtrees, one of which contains z as its separator. We have $T^*(v) = T$, $T^*(w) = T^1(v)$ and $T^*(u) = T^2(v)$

Let T^{subtrees} be the collection of all subtrees obtained by the resulting partitioning, on all levels of the recursion. Note that the trees on each level are disjoint but the entire collection contains overlapping trees. Moreover, in this partitioning, each vertex v in T belongs to a unique subtree $T_l(v) \in T^{\text{subtrees}}$ on each level l of the recursion, up to the level $l(v)$ in which v itself is selected as the separator. The subtrees $T = T_1(v), T_2(v), \ldots, T_{l(v)}(v)$ are referred to as the *ancestor subtrees* of v. Define the *separator tree* T^{sep} to be the tree rooted at the level-1 separator of T, with the level-2 separators as its children, and generally, with each level $j+1$ separator as the child of the level-j separator above it in the decomposition. For a vertex v in T, let $s_j(v)$ denote the *level-j separator of* v, i.e., the ancestor of v in T^{sep} at depth j. We associate each vertex v with the subtree $T^*(v) = T_{l(v)}(v)$ for which v is chosen as its separator. If v is a level-j separator, then $T^*(v)$ is referred to as a *level-j subtree* (see Fig. 1).

For $1/2 \leq \delta < 1$, a *δ-separator* of T is a vertex v whose removal breaks T into disconnected subtrees of at most $\delta|T|$ vertices each. It is a well known fact that every tree has a δ-separator (even for $\delta = 1/2$), and that one can recursively partition the tree by δ-separators. Such a decomposition is termed *δ-separator decomposition*. It is easy to see that the depth of the corresponding separator tree T^{sep} is $O(\log|T|)$. In the special case where $\delta = 1/2$, the separator vertex is called a *perfect separator*, and the decomposition is called a *perfect separator decomposition*.

Representations for separator decompositions: One may define a distributed representation for separator decompositions in trees in various ways. For our purposes, we define a *separator decomposition representation* as follows. Each vertex v in a tree T is given a label $L(v)$ so that the following hold.

1. Each vertex has a unique label, i.e., $L(u) \neq L(v)$ for every two vertices $u, v \in T$.
2. Given the label $L(v)$ of some vertex v and an integer $1 \leq i \leq l(v)$, one can extract the label $L(u)$ where u is the level-i separator of v.

Note that by the first requirement, the maximum number of bits in a label in an n-vertex tree is $\Omega(\log n)$ for any separator decomposition representation.

The supported functions: We consider labeling schemes for supporting the following functions F on pairs of vertices u, v of a rooted tree.

(a) **Routing:** $F(u, v)$ is the port number at u leading to the next vertex on the (shortest) path from u to v.
(b) **Ancestry relation:** $F(u, v) = 1$ if u is an ancestor of v in the tree, else $F(u, v) = 0$.
(c) **NCA relation:** assuming each vertex z has a unique identifier $id(z)$ (encoded using $O(\log n)$ bits), $F(u, v)$ is the identifier $id(z)$ of the nearest common ancestor (NCA) z of u and v, i.e., the common ancestor of u and v of maximum depth.

Labeling schemes: An *F-labeling scheme* $\pi = \langle \mathcal{M}_\pi, \mathcal{D}_\pi \rangle$ is composed of the following components:

1. A *marker* algorithm $\mathcal{M}_\pi$ that given a tree, assigns a label $L(v)$ to each vertex v in the tree.
2. A polynomial time *decoder* algorithm $\mathcal{D}_\pi$ that given the labels $L(u)$ and $L(v)$ of two vertices u and v in the tree, outputs $F(u, v)$.

We note that in our schemes, the labels given to the vertices may contain several fields. In order to distinguish between the different fields of some label one can use an additional label $L'(v)$ for v, which has the same number of bits as $L(v)$ and whose 1's mark the locations where the fields of $L(v)$ begin. Clearly, adding $L'(v)$ does not increase the asymptotic label size.

The dynamic models: The following types of topology changes are considered.

Add-leaf: A new vertex u is *added* as a child of an existing vertex v.

Remove-leaf: A leaf u of a tree is *deleted*.

Subsequent to a topology change, both relevant vertices u and v are informed of it. When a new edge is attached to a vertex v, the corresponding port at v is assigned (either by an adversary or by v) a unique port-number (i.e., at any time, the port numbers at v are distinct), encoded using $O(\log n)$ bits.

All of the results in this paper, except for those on routing, concern the *weak adversary model*, in which an adversary can freely select and change the port numbers at any vertex (as long as they remain disjoint at that vertex). Our dynamic routing schemes consider the following two port models. In the *designer port model*, each vertex v is allowed to freely select and change the port numbers on its incident ports at any time (as long as they remain disjoint), while in the *adversary port model*, the port numbers at each vertex are fixed by an adversary (once the adversary assigns a port number, the number remains unchanged).

Various dynamic models are considered in the literature. In the *leaf-increasing* tree model, cf. [4,14,19], the only topology change allowed is that of a leaf joining the tree, and in the more general *leaf-dynamic* tree model, cf. [3,14,19], leaves can either be added to or removed from the tree. All the results in this paper apply for the leaf-dynamic tree model.

In this paper, we consider the *controlled model* (considered also in [3,16,18], see [16] for more details and motivations). In this model, prior to a topological change at vertex v, a *request* to perform it arrives at v. The arrival of the request triggers the activation of an update protocol $\mathcal{U}$ at v, which is allowed to exchange messages over edges of the underlying graph. The update protocol maintains the labels of the vertices to fit the requirements of the corresponding problem. In addition, the update protocol must grant a *permit* to the request at v, after finite time. Vertex v implements the topological change only after the corresponding request is granted a permit from the update protocol.

In the leaf-increasing model (where no deletions occur), our dynamic schemes can operate under the weak *uncontrolled* model, in which the topological changes (insertions of leaves) may occur spontaneously, without being delayed by the update protocol.

Our schemes are required to be correct only at quiet times, i.e., times for which all necessary updates concerning the previous topological changes have occurred. It can easily be shown that no dynamic separator decomposition scheme can be expected to operate correctly also in non-quiet times. For example, if s is a separator of some level of some vertex u, then if the label of s changes at some time t, then the label $L(u)$ of u must also change at the same time t, since given the label $L(u)$ one should be able to extract $L(s)$.

For a static scheme π on n-vertex trees, we are interested in the following complexity measures.

The *label size*, $\mathcal{LS}(\pi, n)$, is the maximum number of bits in $L(v)$ taken over any vertex v.

The *message complexity*, $\mathcal{MC}(\pi, n)$, is the maximum number of messages (of size $O(\log n)$) sent by a distribute algorithm assigning the labels of π.

In the leaf-dynamic tree model, where both additions and deletions of vertices are allowed, instead of measuring the message complexity in terms of the maximal number of vertices in the scenario, we use more explicit time references employing the notation $\bar{n} = (n_1, n_2, \ldots, n_t)$ where, for each $1 \leq j \leq t$, n_j is the size of the tree immediately after the j'th topology event takes place. The definition of $\mathcal{LS}(\pi, n)$ remains as before, and the definition of the message complexity changes into the following.

Message Complexity, $\mathcal{MC}(\pi, \bar{n})$: the maximum number of messages sent by $\mathcal{U}$ during the labeling process in any scenario where n_j is the size of the tree immediately after the j'th topological event takes place.

3 The static separator representation scheme π_{Stat_Sep}

Let us first note that a static compact separator representation scheme is implicitly described in [12]. However, we were not able to extend that scheme to the dynamic scenario. Instead, in this section we present a new static compact separator decomposition representation scheme π_{Stat_Sep} (which is in some sense a relaxation of the scheme in [12]), which we find easier to extend to the dynamic scenario. Scheme π_{Stat_Sep} enjoys label size $\Theta(\log n)$ and message complexity $O(n \log n)$.

3.1 Informal description

Recall that in a δ-separator decomposition of the tree T, each vertex v is a separator of some level. Given a δ-separator decomposition, a simple way of constructing a representation for it is to assign each vertex a disjoint identity and then to label each vertex by the list of identities of v's ancestors in T^{sep}. However, this simple scheme has label size $O(\log^2 n)$. In order to reduce the label size to $O(\log n)$ we exploit the liberty of choosing the labels of the separators. As in the simple scheme described above, our marker algorithm assigns each vertex v a different label $L^{sep}(v)$ containing $l(v)$ fields. However, in contrast to the simple scheme mentioned above, for any $1 < l \leq l(v)$, the l'th field $L^{sep}_l(v)$ of $L^{sep}(v)$ does not contain the identity of the level-l separator of v. Instead, it contains the binary representation of a number proportionate to $|T_{l-1}(v)|/|T_l(v)|$. Moreover, the label of the level-k separator of v is the prefix of $L^{sep}(v)$ containing the first k fields in $L^{sep}(v)$. Informally, these properties are achieved in the following manner. Define the labels $L^{sep}(v)$ of the separators v by induction on their level. The label of the level-1 separator is set to be $\langle 0 \rangle$. Assume that we have defined the labels of all the level-$(l-1)$ separators. For each level-$(l-1)$ separator v, we now define the labels of its children $v_1, v_2, \ldots$ in T^{sep} as follows. For each k, v_k is first assigned a unique number $\rho(v_k)$ (in the sense that if $k \neq k'$ then $\rho(v_k) \neq \rho(v_{k'})$), such that $\rho(v_k) \in [2^{q+3}, 2^{q+4})$ iff $|T^*(v)|/2^{q+1} < |T^*(v_k)| \leq |T^*(v)|/2^q$, or in other words, $\rho(v_k) \in I_q$ iff $|T^*(v_k)| \in J_q(|T^*(v)|)$.

Note that for each q, there could be at most 2^{q+1} children v_k of v in T^{sep} such that $|T^*(v)|/2^{q+1} < |T^*(v_k)| \leq |T^*(v)|/2^q$. Therefore, the interval $I_q = [2^{q+3}, 2^{q+4})$ contains sufficiently many integers so that every separator v_k satisfying $|T^*(v_k)| \in J_q(|T^*(v)|)$ can be issued a distinct integer in I_q.

For every k, after assigning the vertex v_k a number $\rho(v_k)$ as described above, the label of v_k is set to be the concatenation $L^{sep}(v_k) = L^{sep}(v) \circ \rho(v_k)$. The fact that the labels are disjoint follows from the fact that for each k, $\rho(v_k)$ is unique. Note that the label of a level-l separator u can be considered as a sequence of l fields $L^{sep}(u) = L^{sep}_1(u) \circ \cdots \circ L^{sep}_l(u)$. Moreover, for each $1 \leq j < l$, the label of the level-j separator of u is simply $L^{sep}_1(u) \circ \cdots \circ L^{sep}_j(u)$. In addition, for $1 \leq j \leq l$, the $j+1$'st field $L^{sep}_{j+1}(u)$ is proportionate to $|T_j(u)|/|T_{j+1}(u)|$. This property is used to show that the label size is $O(\log n)$.

In order to implement the scheme π_{Stat_Sep} by a distributed protocol, when the separator v wishes to assign a unique value $\rho(v_k) \in I_q$ to one of its children (in T^{sep}), it somehow needs to know which values it had already assigned in the range I_q. For this purpose, for every $-1 \leq q \leq \lceil \log n \rceil$, v maintains a counter $c_q(v)$ counting the number of values $\rho(v_k) \in I_q$ that were already assigned by it. Whenever v wishes to assign a new value $\rho(v_k) \in I_q$, it simply selects $2^{q+3} + c_q(v)$ and then raises $c_q(v)$ by 1. The fact that $\rho(v_k)$ indeed belongs to I_q is ensured by the following invariant, which holds throughout the execution at every vertex v.

Counters invariant at v:

For every $-1 \leq q \leq \lceil \log n \rceil$, the set of currently assigned values in I_q is a prefix of I_q, namely, $[2^{q+3}, 2^{q+3} + c_q(v) - 1]$.

The description of Protocol STAT_SEP(T), which assigns each vertex v the label $L^{sep}(v)$, is rather straightforward. However, we still prefer to describe it formally, in order to ease the understanding of the more involved Protocol SHUFFLE presented later on.

3.2 Protocol STAT_SEP(T)

We now give a formal description and analysis of the distributed Protocol STAT_SEP(T), which is initiated at the root of a given tree T and assigns each vertex v the label $L^{sep}(v)$.

Before starting the protocol we initialized all labels to be $\langle 0 \rangle$. For each vertex v of level $l(v)$, the recursive Protocol STAT_SEP(T) assigns v the $l+1$'st field in its label, during the l'th level of the recursion, for every $1 \leq l < l(v)$. Thus,

the label of v which consists of $l(v)$ fields (the first field is 0) is assigned during the first $l(v) - 1$ recursive calls to the protocol.

We would like to point out that Protocol SHUFFLE which is described later on, invokes Protocol STAT_SEP on different proper subtrees T' of T. When that protocol calls Protocol STAT_SEP, the vertices in the proper subtree T' may already have labels (that contain certain fields). In this case, the labels in T' are not initialized to $\langle 0 \rangle$, and the recursive protocol Protocol STAT_SEP is executed 'on top' of the existing labels in T', i.e., the output label of the protocol at a vertex u is concatenated with the initial label at u.

On the l'th level of the recursion, Protocol STAT_SEP(T) operates on the subtrees $T^*(u)$ which correspond to the level-l separators u. Let v be a level-l separator of T and let $T^1(v), T^2(v), \ldots$ be the subtrees formed by v. For every i, let v_i be the separator of $T^i(v)$. Note that the vertices $v_1, v_2, \ldots$ are precisely v's children in T^{sep}. On the l'th level of the recursion, v sets the $l+1$'st field of the labels in each $T^k(v)$ to be the number $\rho(v_k)$ assigned to v_k by v according to the method described earlier. Since this value $\rho(v_k)$ of the $l+1$'st field of the labels of all the vertices $T^k(v)$ is the same, we may also refer to this value as $\rho(T^k(v))$.

Before giving the formal description of Protocol STAT_SEP, we first define a *δ-heavychild* decomposition of T, for $1/2 \leq \delta < 1$. In such a decomposition, each non-leaf vertex v marks a *heavy edge*, i.e., an edge leading to one of its children, $h(v)$, such that every other child u satisfies $\omega(u) \leq \delta \cdot \omega(v)$. A *heavy* path is a path consisting solely of heavy edges.

We now give the formal description of Protocol STAT_SEP, starting by describing its Sub-protocols FIND_SEP and SORT_WEIGHT. Sub-protocol FIND_SEP(T) is the straightforward protocol for computing a perfect separator. It is initialized at the root r of an n-vertex tree T and its output is a perfect separator of T. In addition, following the execution of this sub-protocol, each vertex v knows its weight $\omega(v)$. The distributed implementation of the protocol employs standard broadcast and convergecast techniques for disseminating information from the root and collecting weight counts from the leaves upwards toward the root (cf. [20]).

Sub-protocol FIND_SEP(T)

1. The root r broadcasts a signal instructing all the tree vertices to calculate their exact weight through a convergecast process for collecting weight counts. During the convergecast process, each vertex v keeps a pointer $h(v)$ to its heaviest child, i.e., a child u of v such that no other child of v has more descendants than u.
2. The root initiates a broadcast, informing the vertices about n, the correct number of vertices in the entire tree.
3. The root sends a signal along the heavy path containing it, until it reaches a vertex v (possibly r itself) such that the child $h(v)$ satisfies $\omega(h(v)) < n/2$.
4. The output is v.

The proof of the following claim is straightforward.

Claim Sub-protocol FIND_SEP(T) outputs a perfect separator of T using $O(n)$ messages.

Given a separator v, let $N^*(v)$ denote the set of neighbors of v that belong to $T^*(v)$, and for each $x \in N^*(v)$, let T_x be the subtree formed by v that contains x. (I.e., after removing v, the subtree $T^*(v)$ breaks into subtrees, and T_x is the subtree among these subtrees that contain x.)

We now describe Sub-protocol SORT_WEIGHT(m, v), which is initiated at a separator v of some m-vertex subtree $T^*(v)$ of T. When this sub-protocol terminates, every neighbor $x \in N^*(v)$ of v is assigned a unique number $\rho(x)$ which is proportionate to $m/|T_x|$.

We assume that each vertex $x \in N^*(v)$ initially holds the number $n_x = |T_x|$.

Sub-protocol SORT_WEIGHT(m, v)

1. The initiating separator v sends a signal to each of its neighbors. Upon receiving this signal, each neighboring vertex $x \in N^*(v)$ returns a message to v containing n_x.
2. Upon receiving the value n_x from x, vertex v sets $q(x)$ to be the integer satisfying $n_x \in J_{q(x)}(m)$, returns to x the number $\rho(x) = 2^{q(x)+3} + c_{q(x)}(v)$ and sets $c_{q(x)}(v) = c_{q(x)}(v) + 1$.

Claim For every $1 \leq q \leq \lceil \log n \rceil$, at any time during the execution of the sub-protocol, the following hold.

1. $c_q(v) < 2^{q+1}$,
2. If $x \in N^*(v)$ has already been assigned a number $\rho(x)$, then $\rho(x) \in I_q$ iff $n_x \in J_q(m)$,
3. The counters invariant is satisfied.

Proof We prove the claim by induction on the number of times Step 2 is applied in Sub-protocol SORT_WEIGHT(m, v). Assume by induction that the claim holds after the k'th application of Step 2 and consider the $k+1$'st application of Step 2, in which some vertex $x \in N^*(v)$ satisfies $n_x \in J_{q(x)}(m)$.

By our induction hypothesis and by the second and third parts of the claim we obtain that $c_{q(x)}(v) - 1$ vertices $y \in N^*(v)$ have already been assigned a number $\rho(y) \in I_{q(x)}$, and that for each such vertex y, $n_y \in J_{q(x)}(m)$. Together with x we obtain that there exist at least $c_{q(x)}(v)$ neighbors $w \in N^*(v)$ such that $n_w \in J_{q(x)}(m)$. Since every two neighbors $u, w \in N^*(v)$ satisfy $T_u \cap T_w = \emptyset$, we obtain the first

part of the claim. The second and third parts of the claim follow directly from the first part and from the description of Sub-protocol SORT_WEIGHT(m, v). The claim thus follows by induction. □

The proof of the following claim is straightforward.

Claim $\mathcal{MC}(\text{SORT\_WEIGHT}(n, v)) = O(deg(v))$.

We now describe Protocol STAT_SEP(T). Initially all the labels are identical; for every vertex $v \in T$, $L^{sep}(v) = \langle 0 \rangle$. Protocol STAT_SEP$(T)$ recursively calls Protocol STAT_SEP (T'), where T' is a subtree of T rooted at some vertex s.

Protocol STAT_SEP(T) *(for* T *rooted at* r*)*

1. The root r invokes Sub-protocol FIND_SEP(T), which calculates the value $\omega(v)$ for every vertex v and outputs a perfect separator v of T.
2. The root r broadcasts $\omega(r) = |T|$ to all the tree vertices.
3. Upon receiving $\omega(r)$, every vertex $x \in N^*(v)$ creates the number n_x as follows.
   If x is a child of v in T then it sets $n_x = \omega(x)$ (as calculated by Protocol FIND_SEP),
   If x is a parent of v in T then it sets $n_x = \omega(r) - \omega(x) + 1$.
4. The chosen separator v invokes Sub-protocol SORT_WEIGHT$(\omega(r), v)$, after which every vertex $x \in N^*(v)$ is given a unique number $\rho(x)$.
5. For every vertex $x \in N^*(v)$ and every vertex $w \in T_x$, set $L^{sep}(w) = L^{sep}(w) \circ \rho(x)$.
6. Each vertex $x \in N^*(v)$ recursively invokes Protocol STAT_SEP(T_x) at x.

Figure 2 illustrates an example of the labels assigned by Protocol STAT_SEP on some tree T.

The following lemma follows directly from the description of Protocol STAT_SEP(T) and from the second part of Claim 3.2.

Lemma 1 *After Protocol* STAT_SEP(T) *is invoked, the following properties are satisfied for every* $1 \leq k$ *and every two vertices* v *and* u *in* T.
Static separator properties:

SS1: *If* $u \neq v$ *then* $L^{sep}(u) \neq L^{sep}(v)$,
SS2: *If* u *is the level-*k *separator of* v *then* $L^{sep}(u) = L_1^{sep}(v) \circ L_2^{sep}(v) \circ \cdots \circ L_k^{sep}(v)$,
SS3: *If* $L_{k+1}^{sep}(v) \in I_q$ *then* $|T_{k+1}(v)| \in J_q(|T_k(v)|)$.

The correctness and complexity bounds of Protocol STAT_SEP(T) are described in the following lemma.

Lemma 2 *The labels assigned by Protocol* STAT_SEP(T) *to the vertices of* T *form a separator decomposition representation of* T. *Moreover, Protocol* STAT_SEP(T) *has the following complexities.*

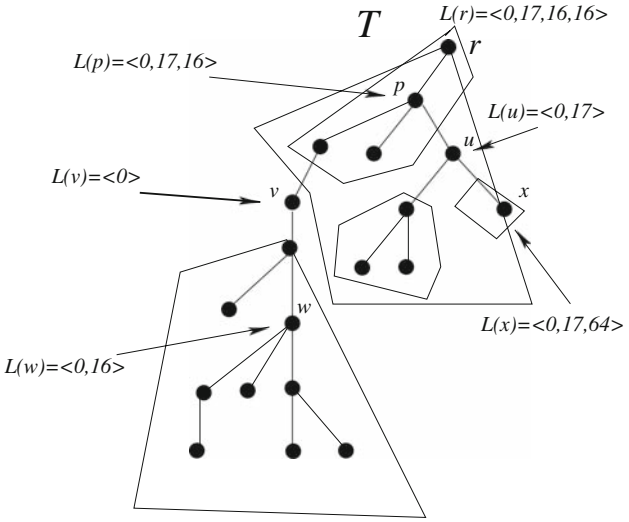

Fig. 2 An example of the labels assigned by Protocol STAT_SEP on some tree T. In the depicted tree T, rooted at r, the vertex v is the level-1 separator of T. Therefore, the label of v is $L(v) = \langle 0 \rangle$. The size of T is 19. Deleting v breaks T into two subtrees, both of size 9; the separator of one of them is w and the separator of the other is u. Assuming that the label of w is assigned before the label of u, we get $L(w) = \langle 0, 2^{3+1} \rangle = \langle 0, 16 \rangle$ and $L(u) = \langle 0, 17 \rangle$. Deleting u from $T^*(u)$ breaks it into three subtrees, of sizes 1, 3 and 4 respectively. The subtree of size 1 formed by u is $T^*(x)$. Since $|T^*(u)| = 9$, the vertex x, which is the separator of $T^*(x)$, is assigned the label $L(x) = L(u) \circ 2^6 = \langle 0, 17, 64 \rangle$. The separator of the subtree of size 4 formed by u is p. Its label is $L(p) = L(u) \circ 2^4 = \langle 0, 17, 16 \rangle$. By removing p, the subtree $T^*(p)$ breaks into three subtrees, each of size 1. One of them is the subtree containing only the vertex $\{r\}$. Since the size of $T^*(p)$ is 4, we get that the label of r is $L(r) = L(p) \circ 2^4 = \langle 0, 17, 16, 16 \rangle$

1. $\mathcal{LS}(\text{STAT\_SEP}, n) = O(\log n)$,
2. $\mathcal{MC}(\text{STAT\_SEP}, n) = O(n \log n)$.

Proof The fact that Protocol STAT_SEP(T) imposes a separator decomposition representation is clear from the static separator properties SS1 and SS2 (which are guaranteed by Lemma 1). We now show that the static property SS3 implies that for every vertex v, the number of bits in $L^{sep}(v)$ is $O(\log n)$.

Fix a level-l vertex v. For every $1 \leq k \leq l$, let $m(k) = |T_k(v)|$ and let $q(k)$ be such that $L_k^{sep}(v) \in I_{q(k)}$. By the static separator property SS3 we obtain that for every $1 < k \leq l, m(k) \leq m(k-1)/2^{q(k)}$. Therefore $\prod_{\{k|1 \leq k \leq l\}} 2^{q(k)} \leq m(1) = n$, yielding $\sum_{\{k|1 \leq k \leq l\}} q(k) \leq \log n$. Since $L_k^{sep}(v) \in I_{q(k)}$, we get that $L_k^{sep}(v)$ can be encoded using $q(k) + 4b$ its and therefore $L^{sep}(v)$ can be encoded using $\sum_{\{k|1 \leq k \leq l\}} (q(k)+4) \leq \log n + 4 \cdot l$ bits. Since the depth of a perfect separator decomposition is $\lceil \log n \rceil$, we obtain that $l = \lceil \log n \rceil$ and the first part of the lemma follows.

Let T^{sep} be the separator tree defined by Protocol STAT_SEP. At the l'th level of the recursion, Protocol STAT_SEP acts on the subtrees $T^*(v)$, which correspond to the

vertices v at depth l in T^{sep}. In particular, Protocol STAT_SEP acts on disjoint subtrees in each level of its recursion. When Protocol STAT_SEP is invoked on some subtree $T^*(v)$, it follows from Claims 3.2 and 3.2 that $O(|T^*(v)|)$ messages are sent by Protocol STAT_SEP. Since the depth of T^{sep} is at most $\lceil \log n \rceil$, the second part of the lemma follows. □

4 Protocol SHUFFLE

4.1 Overview

Protocol SHUFFLE is invoked in the dynamic scenario on subtrees $T' \in T^{subtrees}$ that are suspected to violate some balance properties required in order to maintain the compact separator decomposition on the entire tree T. The goal of Protocol SHUFFLE(T') is to recompute a separator decomposition representation on T' while keeping it consistent with the global separator decomposition representation on T.

During the dynamic scenario, multiple SHUFFLE protocols may be invoked at the same time. Whenever a vertex is asked to simultaneously participate in more than one Protocol SHUFFLE, it continues to participate only in the SHUFFLE protocol, operating on the lowest level subtree among them, and the data structure at that vertex corresponding to the higher level SHUFFLE Protocol is erased. (Observe that this lowest level subtree contains all the higher level subtrees.)

We note that, as shown later, the main Protocol DYN_SEP ensures that if, during the time period that some Protocol SHUFFLE(T') operates, no other Protocol SHUFFLE operates on a subtree containing T', then the subtree T' remains fixed (i.e., no topology change occurs in T') throughout the operation of Protocol SHUFFLE(T').

We assume that each separator v keeps $\omega_0^*(v)$, the number of vertices in $T^*(v)$ after the last application of Protocol SHUFFLE on a subtree containing $T^*(v)$. In addition, we assume that all vertices know some upper bound N on the number of vertices in T. (The value of N may change from time to time as described in the next section.)

The correct operation of Protocol SHUFFLE relies on the fact that the following invariants are maintained for every level-l separator v.

The balance invariant:

The number of topology changes that occurred in $T^*(v)$ from the last application of Protocol SHUFFLE on some subtree containing $T^*(v)$ is at most $\lceil \omega_0^*(v)/8 \rceil$.

The growth property:

If Protocol SHUFFLE($T^*(v)$) is invoked at time t, then at least $\lceil \omega_0^*(v)/16 \rceil$ topology changes occurred in $T^*(v)$ from time t' to time t, where t' is the last time prior to t that another execution of Protocol SHUFFLE was completed on some subtree containing $T^*(v)$.

The fact that these properties are indeed maintained is proved in Lemma 6.1.

4.2 Informal description

Let us start with an informal description of Protocol SHUFFLE (T'). If T' is the entire tree T, then Protocol SHUFFLE(T') consists of simply initializing all labels to $\langle 0 \rangle$, and then running Protocol STAT_SEP(T). Otherwise, if $T' \in T^{subtrees}$ is a proper subtree of T, then let v be its *forming separator*, i.e., the separator whose removal from the subtree $T^*(v)$ formed T'. Let l be the level of v and let $T^1(v)$, $T^2(v)$, ... be the subtrees formed by v. Without loss of generality assume $T' = T^1(v)$.

As mentioned, the goal of Protocol SHUFFLE(T') is to recompute a separator decomposition representation on T' while keeping it consistent with the global separator decomposition representation on T. One way of achieving this goal is by simply initializing the labels in $T^*(v)$ to $L^{sep}(v)$ and then running Protocol STAT_SEP on $T^*(v)$. However, this method would yield undesirably large message complexity, which relates to the size of $T^*(v)$ rather than to the size of T'. Instead, we propose a slightly more complicated method, which yields only $O(|T'| \log |T'|)$ message complexity.

Protocol SHUFFLE(T') is conceptually composed of three stages. In the first stage, all the labels in T' are initialized to be $L^{sep}(v)$ (which contains l fields). At the second stage, the $l+1$'st field of the labels in T', $\rho(T')$, is initialized so that it is proportionate to $\omega_0^*(v)/|T'|$ and distinct from $\rho(T^i)$ for every $i > 1$. At the third stage, Protocol STAT_SEP is invoked on T' to initialize the subsequent ($l+2$'nd, $l+3$'rd, etc) fields of the labels in T' according to a perfect separator decomposition of T'.

Note that as $T^*(v)$ is of level l, the subtree T' is of level $l+1$, and when Protocol STAT_SEP is called on T' in the third stage of Protocol SHUFFLE(T'), the labels in T' are already assigned $l+1$ fields. This is consistent with the case that T' is the whole tree, in the sense that the whole tree is a subtree of level 1, and before calling Protocol STAT_SEP(T), the labels are initialized to contain a single field, namely, $\langle 0 \rangle$. Thus, no matter whether T' is a proper subtree of T or whether T' is T itself, when we call Protocol STAT_SEP(T'), the labels in T' are already assigned l' fields, where l' is the level of the subtree T'. Moreover, observe that the label of the separator of T' (which is of level l') does not change during the application of Protocol STAT_SEP(T'), and that the label of any other vertex u in T' (which contains $l(u) > l'$ fields) is assigned during the first $l(u) - 1$ recursive levels of Protocol STAT_SEP(T').

It is relatively easy to implement the first and third stages of Protocol SHUFFLE(T'). Let us now describe informally how Protocol SHUFFLE implements the second stage. In order for the new assigned value $\rho_{new}(T')$ to be proportionate to

$\omega_0^*(v)/|T'|$, it may need to be in some different interval I_q than before. We use the counters $c_q(v)$ (described in the previous section) to count the number of values in I_q that were already assigned. When vertex v wishes to assign T' a new value $\rho_{new}(T') \in I_q$, it selects the value $2^{q+3} + c_q(v)$ and then raises $c_q(v)$ by 1. However, in contrast to Protocol STAT_SEP, the counters invariant is not necessarily maintained. Instead, the fact that $2^{q+3} + c_q(v) \in I_q$ results from the following more involved argument. Let $\mathcal{S}$ be the last execution of Protocol SHUFFLE on a subtree containing v. After applying $\mathcal{S}$, $c_q(v)$ was relatively small. Let T'' be one of the subtrees $T^i(v)$ that received a new value in I_q after $\mathcal{S}$ was invoked and let $\mathcal{S}''$ be the SHUFFLE protocol applied on T'' after which T'' received this value. By combining the balance invariant (for the separator of T'') with the growth property (for T''), we obtain that the number of topology changes that occurred in T'' from the time $\mathcal{S}$ was invoked until the time $\mathcal{S}''$ was invoked is proportionate to $\omega_0^*(v)/2^q$. On the other hand, by the balance invariant, the total number of vertices joining $T^*(v)$ from the time $\mathcal{S}$ was invoked is at most $\omega_0^*(v)/8$. Combining these two observations, we obtain that the number of subtrees that received a new value in I_q after $\mathcal{S}$ was invoked is small enough to guarantee that $2^{q+3} + c_q(v) \in I_q$.

4.3 Formal description

We now give a formal description of Protocol SHUFFLE(T') by first describing Sub-protocol INIT(T'). We distinguish between two cases. The first case is when T' is the entire tree T. In this case, Sub-protocol INIT(T) is initiated at the root (as is Protocol SHUFFLE(T)), and for every vertex v, it merely initializes its label $L^{sep}(v)$ to be $\langle 0 \rangle$, and sets $c_q(v)$, for every $-1 \leq q \leq \lceil \log N \rceil$, to be the counter c_q at v (which is also used by Protocol STAT_SEP). The other case is when $T' \in T^{subtrees}$ and T' is a proper subtree of T. In this case, let v be the forming separator of T', and let l be its level. Recall, T' was formed after removing v from the subtree $T^*(v)$. In particular, this implies that T' contains a single neighbor of v. Let $x(T')$ denote this neighbor of v. Sub-protocol INIT(T') is initiated at $x(T')$ (as is Protocol SHUFFLE(T')). Throughout this subsection we consider the proper subtree T' as rooted in $x(T')$.

Protocol INIT(T')

1. The vertex $x(T')$ initiating the protocol calculates $|T'|$ through a convergecast operation on T', and delivers a message to v containing the value $|T'|$.
   Let q be the integer satisfying $|T'| \in J_q(\omega_0^*(v))$. (Note that such $-1 \leq q \leq \lceil \log N \rceil$ must exist since by the balance invariant, $|T'| \leq |T^*(v)| < 2 \cdot \omega_0^*(v)$.)
2. The separator v broadcasts a message containing $\langle L^{sep}(v), \rho(T') \rangle$, where $\rho(T') = 2^{q+3} + c_q(v)$, to every vertex $u \in T'$.
3. Every vertex $u \in T'$ sets $L^{sep}(u) = L^{sep}(v) \circ \rho(T')$, and sets $c_q(u) = 0$ for every $-1 \leq q \leq \lceil \log N \rceil$.
4. Vertex v updates its counter $c_q(v)$ to be $c_q(v) + 1$.

Let x' be the initiator of Protocol SHUFFLE(T'). As mentioned before, this initiator is the root if T' is the entire tree and $x(T')$ otherwise. The description of Protocol SHUFFLE (T') now becomes very simple.

Protocol SHUFFLE(T')

1. The initiator x' invokes INIT(T').
2. Next, x' invokes Protocol STAT_SEP(T') (recall, in this application of Protocol STAT_SEP(T'), we consider T' as rooted at x').

Since Protocol INIT(T') incurs $O(|T'|)$ messages, we obtain the following lemma using Lemma 2.

Lemma 3 $\mathcal{MC}(\text{SHUFFLE}(T')) = O(|T'| \log |T'|)$.

Lemma 4 *Assume that the balance invariant and the growth property are maintained throughout the dynamic scenario. Let T' be one of the subtrees formed by a level-l separator v. Consider some application of Protocol* SHUFFLE(T') *and assume that during its operation, no other Protocol* SHUFFLE *operates on a subtree strictly containing T', and that no topology changes occur in T'. Then, after the application of Protocol* SHUFFLE(T')*, the following* SHUFFLE *properties are satisfied for every $k \geq l$ and every two vertices u and w in T'.*

SHUFFLE **properties:**

- **Sh1:** *If $u \neq w$ then $L^{sep}(u) \neq L^{sep}(w)$.*
- **Sh2:** *If u is the level-k separator of w then $L^{sep}(u) = L_1^{sep}(w) \circ L_2^{sep}(w) \circ \cdots \circ L_k^{sep}(w)$.*
- **Sh3:** *For every vertex $w \in T'$, if $L_{l+1}^{sep}(w) \in I_q$ for some q then $|T'| \in J_q(|\omega_0^*(v)|)$.*
- **Sh4:** *For $k > l$, if $L_{k+1}^{sep}(u) \in I_q$ then $|T_{k+1}(u)| \in J_q(|T_k(u)|)$.*

Proof Consider some execution of Protocol SHUFFLE(T') as in the lemma. Since no other protocol was operating on a subtree containing T' during this execution, the labels of the vertices in T' may have changed only due to Protocol SHUFFLE(T'). We also assume that the number of vertices in T' remains the same, hence after the application of Protocol SHUFFLE(T'), the SHUFFLE properties Sh1, Sh2 and Sh4 are clearly satisfied by Step 3 of Protocol INIT(T'), Step 2 of Protocol SHUFFLE(T') and Lemma 1. Let us now show that the

SHUFFLE property Sh3 holds as well. Fix some $w \in T'$ and let t be the time that some Protocol SHUFFLE(T') assigned w the $l+1$'st field in its label $L^{sep}(w)$. By Steps 1 and 2 of Protocol INIT(T'), $L^{sep}_{l+1}(w) = \rho(T') = 2^{q+3} + c_q(v)$, where q is such that $|T'| \in J_q(\omega^*_0(v))$ at time t. It is therefore left to prove that at time t, $2^{q+3} + c_q(v) \in I_q$.

Let t_0 be the last time before time t that some protocol SHUFFLE completed its operation on a subtree containing $T^*(v)$. By the first part of Claim 3.2, at time t_0, the counter $c_q(v)$ satisfies $c_q(v) < 2^{q+1}$. Let Γ be the set of SHUFFLE protocols that were invoked between time t_0 and time t on a subtrees $T^i(v)$ formed by v, and satisfying $|T^i(v)| \in J_q(\omega^*_0(v))$ at the time the protocol took place. For every $\mathcal{S} \in \Gamma$, let $T(\mathcal{S})$ denote the subtree on which $\mathcal{S}$ was invoked. Note that between time t_0 and time t, no Protocol SHUFFLE was operating on a subtree containing v. Therefore, between time t_0 and time t, the value $c_q(v)$ can only change due to an application of some protocol SHUFFLE $\in \Gamma$, which increases the value $c_q(v)$ by 1 (this happens in Step 4 of Protocol INIT($T(\mathcal{S})$)). Since at time t_0, the counter $c_q(v)$ satisfies $c_q(v) < 2^{q+1}$, it is enough to show that $|\Gamma| \leq 2^{q+3} - 2^{q+1} = 6 \cdot 2^q$.

Fix some $\mathcal{S} \in \Gamma$. Let t^s be the time in which $\mathcal{S}$ was invoked and let t^s_0 be the last time before t^s that some other SHUFFLE protocol was completed on a subtree containing $T(\mathcal{S})$. Clearly $t_0 \leq t^s_0 \leq t^s \leq t$. Let $\phi(\mathcal{S})$ be the number of topology changes that occurred in $T(\mathcal{S})$ from time t^s_0 until time t^s. Let ω^s (respectively, ω^s_0) be the number of vertices in $T(\mathcal{S})$ at time t^s (resp., t^s_0). Note that since $\mathcal{S} \in \Gamma$, we have that $\omega^s \in J_q(\omega^*_0(v))$, and in particular, $\frac{\omega^*_0(v)}{2^{q+1}} \leq \omega^s$. By the balance invariant, $\omega^s < \frac{9}{8}\omega^s_0$ and therefore $\frac{8}{9} \cdot \frac{\omega^*_0(v)}{2^{q+1}} < \omega^s_0$. On the other hand, by the growth property, $\phi(\mathcal{S}) \geq \frac{\omega^s_0}{16}$ and therefore, we obtain

$$\phi(\mathcal{S}) \geq \frac{\omega^s_0}{16} \geq \frac{\omega^*_0(v)}{36 \cdot 2^q} .$$

By the balance invariant, we obtain that $\sum_{\mathcal{S} \in \Gamma} \phi(\mathcal{S}) \leq \frac{\omega^*_0(v)}{8}$ and therefore

$$|\Gamma| \cdot \frac{\omega^*_0(v)}{36 \cdot 2^q} \leq \frac{\omega^*_0(v)}{8} .$$

This yields that $|\Gamma| \leq \frac{36}{8} \cdot 2^q < 6 \cdot 2^q$ as desired. The lemma follows. □

5 Controllers and reset protocols

As mentioned before, Protocol DYN_SEP invokes Protocol SHUFFLE($T^*(v)$) whenever the number of topology changes in $T^*(v)$ becomes proportionate to the size of $T^*(v)$. For this purpose, we make use of a variant of the $\langle M, W\rangle$-controllers from [16], which allows us to control and estimate the number of topology changes in the subtrees $T^*(v)$.

$\langle M, W\rangle$-controllers and κ-controllers

The $\langle M, W\rangle$-controller of [16] is a distributed protocol that can operate in the leaf-dynamic model. The input to the controller comes in form of a sequence of *requests* arriving online at different vertices. In our setting, each request is for implementing a topology change at a vertex. After a finite time, from the time a request arrives, the controller answers the request by assigning it a *permit*. Each permit may be assigned to at most one request. A topology change at a vertex u takes place only after the corresponding request at u receives a permit from the controller protocol. Moreover, it is assumed that the topology change indeed occurs eventually, after the corresponding request receives a permit.

We would like to point out that the requests arriving at a node w are handled by the controller one by one, in the order of their arrival to w. If a request τ is being handled by the controller, and has not been granted a permit yet, then all subsequent requests arriving at w are put in a queue at w. When τ is finally granted a permit, the requests from the queue are dequeued one by one, according to the First In First Out discipline, and are then handled by the controller.

In one variation, the $\langle M, W\rangle$-cController of [16] may *terminate* at some time t, which implies in particular, that no request is granted a permit after time t, and hence no topology change occurs after that time. The $\langle M, W\rangle$-controller guarantees the following, for every sequence of requests.

- At most M permits are given to requests, and therefore at most M topology changes occur.
- If the controller terminates at some time t, then at least $M - W$ topology changes have occurred by that time. On the other hand, if the controller does not terminate eventually, then every requested topology change eventually occurs.

Note that the above conditions imply that if the controller does not terminate eventually, then the scenario of requests consists of at most M requests. The message complexity of the $\langle M, W\rangle$-controller is $O(n_0 \log^2 n_0 \cdot \log \frac{M}{W+1}) + O\left(\sum_j \log^2 n_j \cdot \log \frac{M}{W+1}\right)$. In particular, if $M = O(W)$ and $M = O(n_0)$ then the message complexity is $O(n_0 \log^2 n_0)$.

The controllers used in this paper are almost always $\langle \lceil \kappa \rceil, \lfloor \kappa/2 \rfloor \rangle$-controllers for real $\kappa > 0$, hereafter referred to simply as κ-controllers. In particular, for a vertex v, let

$$\kappa(v) = \omega^*_0(v)/8 .$$

Protocol DYN_SEP applies a $\kappa(v)$-Controller on every subtree $T^*(v)$. When considering the operation of the $\kappa(v)$-controller on the subtree $T^*(v)$, we assume that this subtree is rooted at v. In particular, a vertex u participates in multiple

controllers, one operating on each ancestor subtree $T_i(u)$ of u. The permits issued by a controller operating on a level-i subtree are called *level-i permits*. In Protocol DYN_SEP, a topology change at a vertex u of level $l(u)$ occurs only after receiving a level-i permit for each $1 \leq i \leq l(u)$.

In fact, the $\kappa(v)$-controller used by Protocol DYN_SEP is a slightly modified variant of the controller of [16] mentioned above. Let us first note that just before the controller of [16] terminates, it performs a downcast and upcast operation, for making sure that all the permits that were supposed to be delivered by the controller to requests reach their destination, and that subsequently, all corresponding topology changes occur. Since in Protocol DYN_SEP a topology change may occur only after receiving a level-i permit for each $1 \leq i \leq l(u)$, we let the $\kappa(v)$-controller employ a slightly modified version of this downcast and upcast operation. Specifically, the operation is modified to guarantee the following conditions.

- All the level-l permits that were supposed to be delivered by the $\kappa(v)$-controller to requests, reach their destination. Moreover, if some vertex u received the level-l permit, and has not yet received some level-i permit for $l(u) \geq i > l$, then the broadcast operation delivers it also a level-i permit.
- All the topology changes corresponding to requests in $T^*(v)$ that were granted a level-l permit, indeed occur. Note that for this condition to hold, the broadcast may need to wait for a level-i permit to be delivered to the request, for every $1 \leq i < l$.

The controller terminates when the upcast is completed (and therefore the above conditions hold). The following claim follows.

Claim Using $O(\omega_0^*(v) \log^2 \omega_0^*(v))$ messages, the $\kappa(v)$-controller, operating on $T^*(v)$, guarantees the following, for the time period it operates.

- At most $\lceil \kappa(v) \rceil$ topology changes occur in $T^*(v)$,
- If the $\kappa(v)$-controller terminates, then the number of topology changes that occurred in $T^*(v)$ is at least $\lceil \kappa(v)/2 \rceil$.

The RESET protocol

Note that it might happen that a controller starts to handle a request, but does not grant it a permit eventually (since it terminates before granting the request). We refer to such requests as *unanswered*.

As described later, after the controller on $T^*(v)$ terminates, Protocol SHUFFLE is invoked on $T^*(v)$. When the SHUFFLE protocol is completed, $T^*(v)$ is decomposed according to a new separator decomposition. In many cases, a new controller will then start on each of the decomposed subtrees in $T^*(v)$ (including, in particular, on $T^*(v)$ itself). Therefore, each vertex $w \in T^*(v)$ will start participating in a new controller on $T_j(w)$, for every level $l(w) \leq j \leq l(v)$. If there exists an unanswered request τ at the node w at the time it starts participating in a new controller, then with regard to this controller, node w starts handling the request τ from scratch.

Note that we must still guarantee that whenever a topology change is requested, it will indeed occur eventually. Potentially, the unanswered request τ, which was not granted a permit by the previous controller on $T_j(w)$, might again fail to receive an answer from the new controller on $T_j(w)$, due to the asynchronous nature of the setting. If this happens repeatedly, then the request might remain unanswered forever. In order to prevent such a phenomenon from happening, we employ Protocol RESET(T), which is applied whenever Protocol SHUFFLE completes its operation on the entire tree T. The protocol consists of a simple broadcast and upcast operation that starts at the root and enables the occurrence of all the topology changes that correspond to unanswered requests. When Protocol RESET(T) completes its operation, Protocol SHUFFLE is invoked again on the entire tree T.

6 Dynamic separator decomposition

We now describe the main Protocol DYN_SEP, whose goal is to maintain a compact separator decomposition representation in the dynamic tree T. Protocol DYN_SEP occasionally invokes Protocol SHUFFLE on different subtrees. Informally, the correctness of Protocol DYN_SEP depends on the fact that the SHUFFLE properties hold when a SHUFFLE protocol is completed. However, these are only guaranteed assuming that the balance invariant and the growth property are maintained when the SHUFFLE protocols take place. Protocol DYN_SEP guarantees these assumptions by invoking Protocol SHUFFLE($T^*(v)$) whenever the number of topology changes in $T^*(v)$ becomes proportionate to the size of $T^*(v)$.

Let N be twice the size of the tree T after the last application of Protocol SHUFFLE(T) was completed. We assume that every vertex v keeps the value $\omega_0^*(w)$ for each of its separators w. (This assumption can be removed by slightly modifying Protocol SHUFFLE.) In particular, each vertex in T knows N.

Protocol DYN_SEP

1. Initially, Protocol SHUFFLE is invoked on the entire initial tree T.
2. The root broadcasts a signal allowing every vertex u to participate in the $\kappa(v)$-Controller operating on $T^*(v)$, for every separator v of u (for any such controller, $T^*(v)$ is

considered to be rooted at v). A permit issued by a controller operating on a level-l subtree is called a level-l permit. A topology change at vertex u occurs only after receiving a level-i permit for each $1 \leq i \leq l(u)$. Once the topology change occurs, the corresponding level-i permits are consumed.

The requests arriving at a node u are kept in a queue at u. Once a topology change (which corresponds to the last granted request at u) occurs at u, a request from the queue at u is dequeued according to the First In First Out discipline and is handled by all corresponding controllers.

3. Whenever some $\kappa(v)$-Controller on $T^*(v)$ terminates, the following happens.
   (a) Vertex v invokes Protocol SHUFFLE($T^*(v)$). Whenever a vertex is asked to simultaneously participate in more than one Protocol SHUFFLE, it continues to participate only in the SHUFFLE protocol operating on the lowest level subtree among them, and the data structure at that vertex corresponding to the higher level SHUFFLE Protocol is erased. In addition, whenever a vertex u participates in some level-l SHUFFLE protocol, it stops participating in the level-i controllers for any $i > l$, and the corresponding data structure at u is erased.
   (b) Consider some application of Protocol SHUFFLE ($T^*(v)$) which was not applied by Step 1 above. Once Protocol SHUFFLE($T^*(v)$) is completed, the following happens.
   – If $T^*(v)$ is a proper subtree of T, then v broadcasts a signal to all nodes w in $T^*(v)$. When a node $w \in T^*(v)$ receives this signal, the following happen.
   For every separator $u \in T^*(v)$ of w (i.e., for every separator u of w of level j such that $l(w) \geq j \geq l(v)$), node w starts participating in a new $\kappa(u)$-Controller on $T^*(u)$, where $\kappa(u) = \omega_0^*(u)$, for the new value of $\omega_0^*(u)$. (Recall that the new value of $\omega_0^*(u)$ was calculated during the operation of Protocol SHUFFLE($T^*(v)$).)
   When u starts participating in new controllers, then, if there exists an unanswered request at u, then the controllers start handling this request from scratch (and all other requests at u continue to wait in the queue). Otherwise, a request from the queue is dequeued according to the First In First Out discipline, and is then handled by the controllers.
   – Otherwise ($T^*(v) = T$), v invokes Protocol RESET(T) (which enables all topology changes that correspond to unanswered requests to occur). When Protocol RESET(T) is completed, the current tree is considered as the new initial tree and Protocol DYN_SEP is then invoked on this (new) initial tree (in particular, in Step 1 of Protocol DYN_SEP, Protocol SHUFFLE is invoked on the entire tree).
4. Consider the case that a new vertex u is added as a child of a level-l separator v, and that this topology change does not occur as a part of Protocol RESET(T). In this case, u becomes the separator of the new $l+1$'st-level subtree $T_u = \{u\}$. We consider u to be a child of v in the separator decomposition tree T^{sep}. When u joins the tree it invokes Protocol SHUFFLE($\{u\}$) after which u is assigned the label $L^{sep}(u)$. When the protocol is completed, Step 3.b above is applied. (In particular, this means that u invokes the $(1,0)$-Controller on T_u.)

6.1 Correctness

We consider the protocol as running in iterations, which are separated by RESET(T) protocols. More precisely, an iteration starts with the completion of Protocol SHUFFLE on the entire tree T and ends with the completion of the consecutive application of Protocol SHUFFLE(T) (if one exists). The next iteration starts only after the following RESET(T) and SHUFFLE(T) protocols are completed. (The first iteration starts after the first SHUFFLE protocol on the initial tree is completed.)

Recall that correctness is required only for quiet times, and note that between iterations, the system is not quiet (because at any time between iterations, either Protocol RESET(T) or Protocol SHUFFLE(T) is running). Therefore, we only need to prove that the scheme is correct during the quiet times in each iteration.

We say that a vertex v is *quiet* at time t if no Protocol SHUFFLE is operating at that time on a subtree strictly containing $T^*(v)$.

Claim Fix some Protocol SHUFFLE($T^*(v)$) and assume that v remains quiet during the time period $\mathcal{T}$ in which Protocol SHUFFLE($T^*(v)$) operated. Then no topology change occurs in $T^*(v)$ during the time period $\mathcal{T}$.

Proof Since v remains quiet during the time period $\mathcal{T}$, a topology change at a vertex $w \in T^*(v)$ may occur only after the corresponding request at w receives a level-l permit from a controller which is operating on $T^*(v)$ during the time period $\mathcal{T}$. However, at that time period, no controller is operating on $T^*(v)$, and therefore during that time period, no topology change can occur in $T^*(v)$. □

Claim Every invoked protocol RESET is eventually completed, and every invoked Protocol SHUFFLE($T^*(v)$) is either completed eventually or v is not quiet at some time during its operation.

Proof Consider first an invocation of Protocol SHUFFLE ($T^*(v)$), such that v remains quiet throughout its opera-

tion. By the previous claim, the subtree $T^*(v)$ remains fixed throughout the operation of this Protocol SHUFFLE($T^*(v)$). Thus, the protocol is eventually completed.

Consider now an invocation of protocol RESET(T). The only way such a protocol may not be completed, is if vertices are constantly being inserted to the tree. Note, that during the time period in which Protocol RESET operates, only topology changes that correspond to unanswered requests may occur. Moreover, observe that there is at most one unanswered request at each node w. Thus, the number of nodes during the operation of a Protocol RESET(T) is at most twice the number of nodes that were when the protocol was initiated. It follows that the invocation of Protocol RESET is completed eventually. □

Claim Every requested topology change eventually occurs.

Proof Recall that the requests arriving at a node u are handled by the controllers one by one, in the order of their arrival to u. Assume, by contradiction that there exists a request such that the corresponding topology change did not occur eventually. Let τ be the first such request, and let u be the vertex to which τ arrived. It follows from the previous claim, that since all the topology changes that correspond to requests that arrived to u prior to τ have occurred, the request τ was dequeued and handled by the corresponding controllers at some time t.

If the request τ received a level-i permit for each $1 \leq i \leq l(u)$, then the corresponding topology change would have occurred eventually, contradicting our assumption. We therefore get that τ never received a level-i permit for some $1 \leq i \leq l(u)$. Thus, from time t, the request τ was not answered by any controller that was operating on $T_i(u)$. If Protocol RESET(T) was applied after time t, then the corresponding topology change would have occurred since τ is unanswered. Therefore, no Protocol RESET(T) was applied after time t.

Recall that if a controller on $T_i(u)$ was not terminated eventually and was not stopped (in Step 3(a) of Protocol DYN_SEP) by some Protocol SHUFFLE (operating on a subtree containing $T_i(u)$), then all the requests in vertices of $T_i(u)$ eventually receive a level-i permit. It therefore follows that there exists a level j such that the number of controllers that were operating on $T_j(u)$ after time t and were terminated, is infinite. By Claim 5 it follows that the number of topology changes in T is infinite, and therefore, again by Claim 5, the controller that operates on the entire tree T at time t must terminate at some later time. Therefore, by Step 3(b) in Protocol DYN_SEP, Protocol RESET(T) must be applied at least once after time t, contradiction. □

The following claim follows from Claim 5 and from the fact that during the execution of a given iteration, the only cases where Protocol SHUFFLE($T^*(v)$) is applied are immediately after the corresponding $\kappa(v)$-Controller terminates.

Claim Fix an iteration of the protocol. The balance invariant and the growth property are maintained at all times during the execution of the iteration.

Recall that $N/2$ is the size of tree T after the last application of Protocol SHUFFLE(T) was completed. The following claim is obtained by the growth property and the balance invariant (applied on the entire tree T). Recall that n denotes the current number of vertices in T.

Claim During a given iteration, n satisfies $N/4 < n < N$.

By Lemma 4, Claims 6.1, and 6.1, we obtain the following lemma.

Lemma 5 *Consider an application of Protocol* SHUFFLE ($T^*(v)$) *and assume that v remains quiet during its operation. Then after Protocol* SHUFFLE($T^*(v)$) *completes its operation, the* SHUFFLE *properties are satisfied for every $k \geq l$ and for every two vertices u and w in $T^*(v)$.*

Recall that by item 3(a) in Protocol DYN_SEP, whenever a vertex is asked to simultaneously participate in more than one Protocol SHUFFLE, it continues to participate only in the SHUFFLE protocol operating on the lowest level subtree among them. Therefore, if v does not remain quiet during the application of Protocol SHUFFLE($T^*(v)$), then before the next quiet time (in which the whole system is quiet), all nodes in $T^*(v)$ will participate in some Protocol SHUFFLE($T^*(u)$), where $T^*(u)$ strictly contains $T^*(v)$, and u remains quiet during the application of Protocol SHUFFLE($T^*(u)$). By this observation and by the above lemma we obtain the following corollary.

Corollary 1 *At any quiet time, the following properties are satisfied for any two vertices u and w.*

- *If $u \neq w$, then $L^{sep}(u) \neq L^{sep}(w)$,*
- *If u is the level-k separator of w, then $L^{sep}(u) = L_1^{sep}(w) \circ L_2^{sep}(w) \circ \cdots \circ L_k^{sep}(w)$.*

Corollary 2 *At any quiet time, Protocol* DYN_SEP *maintains a separator decomposition representation on T.*

6.2 Analysis

6.2.1 Label size

Before showing that the label size of the scheme is $O(\log n)$, we need the following lemma, which is a relaxation of the static separator property SS3.

Lemma 6 *Fix an iteration, a vertex v and a time t during the iteration, such that no* SHUFFLE *protocol is operating on a subtree containing $T^*(v)$ at time t. Let $T^1(v), T^2(v) \ldots$*

be the subtrees formed by v (at time t). Then, at time t, for every $i \geq 1$ and for every $w \in T^i(v)$, if $L^{sep}_{l+1}(w) \in I_q$ then $|T^i(v)| \in \widehat{J}_q(\omega^*_0(v))$.

Proof Fix some i and let $T' = T^i(v)$. Let w be some vertex in T' and denote by $\mathcal{S}$ the SHUFFLE protocol in which the current value of $L^{sep}_{l+1}(w)$ was assigned. Let $\mathcal{T}(\mathcal{S})$ denote the subtree on which $\mathcal{S}$ was invoked (note that $\mathcal{T}(\mathcal{S})$ contains T'). Let t_0 be the time when $\mathcal{S}$ assigned w its label. Let q be the integer such that $L^{sep}_{l+1}(w) \in I_q$ at time t_0. Note that between time t_0 and time t, the value of $L^{sep}_{l+1}(w)$ remains the same, and therefore, in particular, remains in I_q. Let T'_0 be the subtree T' at time t_0. Let s be the separator of T', i.e, $T' = T^*(s)$.

We now claim that s is quiet throughout the time period $\mathcal{T}(S)$ that $\mathcal{S}$ operates. Otherwise, some Protocol SHUFFLE is operating at some time during $\mathcal{T}(S)$, on a subtree T'' strictly containing T', i.e., $T'' \supseteq T^*(v)$. By the choice of $\mathcal{S}$, Protocol SHUFFLE(T'') assigned w a label after time t. Therefore, Protocol SHUFFLE(T'') was operating in time t, contradicting the assumptions of the lemma. We therefore get that s is quiet throughout the time period $\mathcal{T}(S)$. Therefore, by Lemma 5, the shuffle properties hold on $\mathcal{T}(\mathcal{S})$ when $\mathcal{S}$ completes its operation. Note also, that by Claim 6.1, the number of vertices in $\mathcal{T}(\mathcal{S})$ remains the same throughout the operation of $\mathcal{S}$. We thus have $|T'_0| \in J_q(\omega^*_0(v))$, i.e., $\omega^*_0(v)/2^{q+1} \leq |T'_0| < \omega^*_0(v)/2^q$.

By the balance invariant, at time t we have $\frac{7}{8}|T'_0| \leq |T'| \leq \frac{9}{8}|T'_0|$, implying that

$$\frac{\omega^*_0(v)}{2^{q+1}} \leq |T'_0| \leq \frac{8}{7}|T'| \leq \frac{9}{7} \cdot |T'_0| < 2|T'_0| < \frac{\omega^*_0(v)}{2^{q-1}},$$

and therefore

$$\frac{\omega^*_0(v)}{2^{q+2}} \leq |T'| < \frac{\omega^*_0(v)}{2^{q-1}}.$$

In other words, $|T'| \in \widehat{J}_q(\omega^*_0(v))$ and the lemma follows. □

Lemma 7 $\mathcal{LS}(\text{DYN\_SEP}, n) = O(\log n)$.

Proof Let us first prove that there exists a fixed constant c, such that during any iteration, $|L^{sep}(v)| < c \log n$ for any vertex v and any time t, in which no SHUFFLE protocol is operating on a subtree containing v.

Fix a vertex v and consider such a time t. Let $l = l(v)$. Recall that $T_1(v), T_2(v), \ldots, T_l(v) = \{v\}$ are the ancestor subtrees of v. For every $1 \leq k \leq l$, let $m(k) = |T_k(v)|$ at time t and let $m_0(k)$ be the value of $|T_k(v)|$ after the last SHUFFLE operation took place on a subtree containing $T_k(v)$.

We first show that $m(i) \leq \frac{5}{7}m(i-1)$, for any $i > 1$. Fix $i > 1$, and let $\mathcal{S}$ be the last SHUFFLE protocol that was invoked on a subtree containing $T_{i-1}(v)$ before time t. Note that by the assumption on t, $\mathcal{S}$ was completed before time t. Recall that $T_{i-1}(v)$, the subtree on which $\mathcal{S}$ was applied, was decomposed by $\mathcal{S}$ into a perfect separator decomposition, hence $m_0(i) \leq m_0(i-1)/2$. By the balance invariant, the number of topology changes occurring in T_{i-1} between times t_0 and t is at most $m_0(i-1)/8$, and in particular, $\frac{7}{8} \cdot m_0(i-1) \leq m(i-1)$ and $m(i) - m_0(i) \leq m_0(i-1)/8$ at time t. Consequently,

$$\begin{aligned} m(i) &\leq \frac{1}{8} \cdot m_0(i-1) + m_0(i) \leq \frac{5}{8} \cdot m_0(i-1) \\ &\leq \frac{5}{8} \cdot \frac{8}{7} \cdot m(i-1) = \frac{5}{7} \cdot m(i-1). \end{aligned}$$

It follows that $m(i) \leq \frac{5}{7}m(i-1)$, i.e., $|T_i(v)| \leq \frac{5}{7}|T_{i-1}(v)|$. Since this inequality is satisfied for every $i > 1$, we get that $l(v) < c' \log n$ for some fixed constant c'.

For every $1 \leq k \leq l$, let $q(k)$ be such that $L^{sep}_k(v) \in I_{q(k)}$. By Lemma 6, we have $m(k) < m_0(k-1)/2^{q(k)-1}$ for every $1 < k \leq l$, and by the balance invariant, $m_0(k-1) < 2m(k-1)$. It follows that $m(k) < 2m(k-1)/2^{q(k)-1} = m(k-1)/2^{q(k)-2}$ for every $1 < k \leq l$. Therefore $\prod_{\{k|1\leq k\leq l\}} 2^{q(k)-2} \leq m(1) = n$, yielding $\sum_{\{k|1\leq k\leq l\}}(q(k)-2) \leq \log n$. Since $L^{sep}_k(v) \in I_{q(k)}$, we get that $L^{sep}_k(v)$ can be encoded using $q(k)+4$ bits and therefore $L^{sep}(v)$ can be encoded using $\sum_{\{k|1\leq k\leq l\}}(q(k)+4) = 6 \cdot l + \sum_{\{k|1\leq k\leq l\}}(q(k)-2) \leq 6 \cdot l + \log n$ bits. Since $l(v) < c' \log n$, we obtain that at time t, $|L^{sep}(v)| \leq c \log n$ for $c = 1 + 6c'$. Therefore, during an iteration, at any time t in which no SHUFFLE protocol is operating on a subtree containing v, we have $|L^{sep}(v)| < c \log n$.

Consider now the case that there exist some SHUFFLE protocols that are operating at time t on subtrees containing v, and let $\mathcal{S}$ be the SHUFFLE protocol operating on the largest subtree among them. Note, that if $|L^{sep}(v)| \geq c \log n$ at time t, then $\mathcal{S}$ has not yet assigned v a label by time t. Otherwise an adversary can delay any new SHUFFLE protocol, so that when $\mathcal{S}$ is completed, the value of $L^{sep}(v)$ remains as it was in time t, but no SHUFFLE protocol is operating on a subtree containing v, leading to contradiction.

It follows that if $|L^{sep}(v)| \geq c \log n$ at some time t, then either Protocol RESET(T) is running at time t or that there exists a SHUFFLE protocol that is operating at time t on a subtree containing v, and this SHUFFLE protocol has not assigned v a new label by time t.

Consequently, if at some time t the size of the label at v exceeds $c \log n$, then v simply erases its label. This way, the label size is always $O(\log n)$. Moreover, erasing the label at time t does not effect the correctness of the protocol, since correctness is only required for quiet times, and by the above discussion, at the next quiet time, v will have a new label of the appropriate size. The lemma follows. □

6.2.2 Message complexity

Lemma 8 $\mathcal{MC}(\text{DYN\_SEP}, \bar{n}) = O(n_0 \log^4 n_0) + O\left(\sum_j \log^4 n_j\right)$.

Proof First note, that the number of messages resulting from the different applications of Protocol RESET is bounded from above by the number of messages resulted from the different applications of Protocol SHUFFLE.

Fix an iteration, and let $\mathcal{T}$ be the time period in which the iteration was executed. Let us now bound the number of messages sent during the time period $\mathcal{T}$. Recall first that by Claim 6.1, the value N held at each vertex during this time period satisfies $N/4 < n < N$

There are three types of controllers which are applied during the time period $\mathcal{T}$. The first type consists of controllers that terminated, the second consists of controllers that are active at the end of $\mathcal{T}$, and the third type consists of controllers that were stopped by some Protocol SHUFFLE operating on a lower level subtree (see item 3(a) in Protocol DYN_SEP). Note that when $\mathcal{T}$ is a complete time window, i.e., the second application of Protocol SHUFFLE(T) took place within the time period $\mathcal{T}$, there are no controllers of the second type, since all the controllers that where active when the SHUFFLE(T) protocol took place, were stopped by that protocol and are therefore considered as type three controllers.

Let us first bound the number of messages resulting from the different applications of the controllers of the first type. Fix a level l. A *level-l controller* is a controller operating on a level-l subtree. Each level-l $\kappa(v)$-Controller incurs $O(\omega_0^*(v) \log^2 N)$ messages. By the growth property, the fact that the controller terminated implies that $\Omega(\omega_0^*(v))$ topology changes occurred in $T^*(v)$ during its operation. This gives $O(\log^2 N)$ amortized message complexity per topology change per level. Since there are $O(\log N)$ levels and $O(N)$ topology changes during the time period $\mathcal{T}$, it follows that the total number of messages sent by such controllers during the time period $\mathcal{T}$ is $O(N \log^3 N)$.

Let us now turn to bounding the number of messages resulting from applications of controllers of the second type. Fix a level l and consider the first time t just before the time period $\mathcal{T}$ ended. At time t, different controllers (of the second type) where active on the level-l subtrees. Since each of them was applied on a different subtree (at any given time the level-l subtrees are disjoint), we obtain that the total number of messages resulted from all these controllers is $O(N \log^2 N)$. Since there are $O(\log N)$ levels, were obtain that the total number of messages sent by controllers of the second type during the time period $\mathcal{T}$, is $O(N \log^3 N)$.

Finally, let us now bound the number of messages resulting from controller applications of the third type. In particular, fixing two levels l and l' such that $l' < l$, we consider the messages sent by applications of level-l controllers that were stopped by an application of some Protocol SHUFFLE(T') on a level-l' subtree T'. Fix a Protocol SHUFFLE(T'), where T' is a level-l' subtree. Let t be the time Protocol SHUFFLE(T') was completed, and let t_0 be the time the previous application of a SHUFFLE protocol on a subtree containing T' was completed. The level-l controllers that were stopped by this SHUFFLE protocol operated between time t_0 and time t. These controllers operate on disjoint subtrees, and therefore incur $O(|T'| \log^2 N)$ messages in total. Summing over all levels l such that $l > l'$, we obtain that $O(|T'| \log^3 N)$ messages in total are used by the third type of controllers operating in T' between times t_0 and t. On the other hand, by the growth property (applied on T'), the number of topology changes occurring in T' from time t_0 until time t is $\Omega(|T'|)$. This yields $O(\log^3 N)$ amortized message complexity per topology change, for controllers of the third type that were stopped by a level-l' SHUFFLE protocols. Summing over the $O(\log N)$ levels l' and $O(N)$ topology changes during the time period $\mathcal{T}$, the total number of messages sent by the different controllers during the time period $\mathcal{T}$ is $O(N \log^4 N)$.

Let us now bound the number of messages resulting from the different applications of Protocol SHUFFLE. By Lemma 3, the number of messages resulting from the different applications of the SHUFFLE protocols in the iteration is asymptotically bounded from above by the number of messages used by the first type of controllers, which is $O(N \log^3 N)$. Combined, the total number of messages sent during the time period $\mathcal{T}$ is $O(N \log^4 N)$.

Let $\mathcal{T}_\gamma$ be time period corresponding to the γ'th iteration. Let M_γ be the number of messages sent during $\mathcal{T}_\gamma$ and let N_γ be the number of vertices at the beginning of $\mathcal{T}_\gamma$. By the discussion above, $M_\gamma = O(N_\gamma \log^4 N_\gamma)$.

Let $k \geq 1$ be the number of iterations. For every $1 \leq \gamma \leq k$, the number of messages send during the RESET(T) and SHUFFLE(T) protocols which were invoked just before the γ'th iteration started is bounded by $O(N_\gamma \log N_\gamma)$. We therefore have,

$$\begin{aligned} \mathcal{MC}(\text{DYN\_SEP}, \bar{n}) &= O\left(\sum_{\gamma=1}^{k} M_\gamma\right) \\ &= O\left(\sum_{\gamma=1}^{k} N_\gamma \log^4 N_\gamma\right). \end{aligned} \tag{1}$$

Note that if $k > 1$ then for all $1 \leq \gamma < k$, the number of topology changes occurring during $\mathcal{T}_\gamma$ is $\Omega(N_\gamma)$. Hence $N_\gamma \log^4 N_\gamma = O(\sum_i \log^4 n_{\gamma_i})$ for every $1 \leq \gamma \leq k-1$, where n_{γ_i} is the number of vertices when the i'th topology change during $\mathcal{T}_\gamma$ takes place. It follows that if $k > 1$ then $O(\sum_{\gamma=1}^{k-1} N_\gamma \log^4 N_\gamma) = O(\sum_j \log^4 n_j)$, where n_j is the number of vertices when the j'th topology change takes place. In addition, for $k \geq 1$, we clearly have $N_k \log^4 N_k = O(n_0 \log^4 n_0) + O(\sum_j \log^4 n_j)$. Altogether, we get that

$\sum_{\gamma=1}^{k} N_\gamma \log^4 N_\gamma = O(n_0 \log^4 n_0) + O(\sum_j \log^4 n_j)$, which implies by Equation 1 that

$$\mathcal{MC}(\text{DYN\_SEP}, \bar{n}) = O(n_0 \log^4 n_0) + O(\sum_j \log^4 n_j).$$

The lemma follows. □

By Corollary 2 and Lemmas 7 and 8, we obtain the following theorem.

Theorem 1 *At any quiet time, Protocol* DYN_SEP *maintains a separator decomposition representation. Moreover, Protocol* DYN_SEP *has the following complexities.*

1. $\mathcal{LS}(\text{DYN\_SEP}, n) = O(\log n)$
2. $\mathcal{MC}(\text{DYN\_SEP}, \bar{n}) = O(n_0 \log^4 n_0) + O(\sum_j \log^4 n_j)$.

7 Applications: dynamic labeling schemes for trees

In this section we describe our improved dynamic labeling schemes, all of which use $O(n_0 \log^4 n_0) + O(\sum_j \log^4 n_j)$ message complexity. We begin with describing our dynamic compact ancestry labeling scheme.

7.1 Improved ancestry labeling scheme on dynamic trees

We first introduce a new static compact labeling scheme, $\pi_{Stat\_Anc} = \langle \mathcal{M}_{SA}, \mathcal{D}_{SA} \rangle$, supporting the ancestry relation, and then show how to extend it to obtain Scheme $\pi_{Dyn\_Anc} = \langle \mathcal{M}_{DA}, \mathcal{D}_{DA} \rangle$ for the dynamic setting. Scheme π_{Stat_Anc} uses the separator decomposition representation obtained by Scheme π_{Stat_Sep}. For every two vertices v and u, let $s(v, u)$ denote the NCA of v and u in T^{sep}. Scheme π_{Stat_Anc} is based on the fact that a vertex v is an ancestor of a vertex u iff v is an ancestor of $s(v, u)$ and u is a descendant of $s(v, u)$. The label $L(v)$ given by the marker algorithm $\mathcal{M}_{SA}$ to a vertex v is composed of two sublabels, namely, the *separation sublabel*, $L^{sep}(v)$, and the *relative sublabel* , $L^{rel}(v)$. The separation sublabel $L^{sep}(v)$ is the label given to v by the scheme π_{Stat_Sep}. The relative sublabel $L^{rel}(v)$ is composed of $l(v)$ fields. The j'th field of $L^{rel}(v)$ contains two bits indicating whether $s_j(v)$, the level-j separator of v, is an ancestor of v in T, descendant of v in T or neither.

Given two labels $L(v)$ and $L(u)$, of two vertices v and u, one can extract the level i of $s(v, u)$ using the corresponding separation sublabels and then find whether in T, v is an ancestor of $s(v, u)$ and whether u is a descendant of $s(v, u)$, using the i'th field of the corresponding relative sublabels.

In the dynamic scenario, the separation sublabels are maintained using the dynamic scheme π_{Dyn_Sep}. In order to maintain the relative sublabels, we slightly modify Protocol SHUFFLE so that whenever a vertex v is assigned a new level-j separator, the j'th field in its relative sublabel is updated appropriately, according to whether v is an ancestor, descendant or neither of this separator. By Theorem 1 we therefore obtain the following theorem.

Theorem 2 *Scheme π_{Dyn_Anc} maintains a compact ancestry labeling scheme using $O(n_0 \log^4 n_0) + O(\sum_j \log^4 n_j)$ messages.*

7.2 Improved dynamic routing labeling schemes

We now briefly describe our dynamic routing schemes π_{rout} which have optimal label size up to a multiplicative factor of $O(\log \log n)$. I.e., the label size of π_{rout} is $O(\log n \cdot \log \log n)$ for the designer port model, and $O(\log^2 n)$ for the adversary port model.

Let v be some vertex, and let $l(v)$ be the level for which v was chosen as a separator. For each $1 \leq i \leq l(v)$, let $s_i(v)$ be the i'th separator of v. The label of v given by π_{rout} is composed of three sublabels. The first is the *separator sublabel* $L^{sep}(v)$ which is the label given to v by π_{Dyn_Sep} (recall that $L^{sep}(v)$ contains $l(v)$ fields). The second and third sublabels are the *port-to-separator sublabel* $L^{to-sep}(v)$ and the *port-from-separator sublabel* $L^{from-sep}(v)$. Each of these sublabels also contains $l(v)$ fields. The i'th field in $L^{to-sep}(v)$, namely $L_i^{to-sep}(v)$, is the port number at v, leading from v to the next vertex on the shortest path connecting v and $s_i(v)$. The i'th field in $L^{from-sep}(v)$, namely $L_i^{from-sep}(v)$, is the port number at $s_i(v)$, leading from $s_i(v)$ to the next vertex on the shortest path connecting $s_i(v)$ and v. By slightly modifying Protocol SHUFFLE, we can ensure that whenever Protocol *Dyn_Sep* updates the i'th field in $L^{sep}(v)$, the i'th fields in the sublabels $L^{to-sep}(v)$ and $L^{from-sep}(v)$ are also updated appropriately.

Given the labels $L(u)$ and $L(v)$ of two vertices u and v, the port number at u, leading from u to the next vertex on the shortest path connecting u and v, is determined as follows. If $L^{sep}(u)$ is a prefix of $L^{sep}(v)$ and $L^{sep}(u)$ contains i fields, then $u = s_i(v)$ and therefore the desired port number is $L_i^{from-sep}(v)$. If, on the other hand, $L^{sep}(u)$ is not a prefix of $L^{sep}(v)$ then let i be the last index such that $L_i^{sep}(u) = L_i^{sep}(v)$. In this case, the i'th separator of u, $s_i(u)$, must be on the path connecting u and v and must be different than u. Therefore, the desired port number is $L_i^{to-sep}(u)$.

Scheme π_{rout} is clearly a correct dynamic routing schemes. Let us now analyze its label size. First, for each vertex v, the separator sublabel $L^{sep}(v)$ contains $O(\log n)$ bits. Both the port-to-separator sublabel $L^{to-sep}(v)$ and the port-from-separator sublabel $L^{from-sep}(v)$ contain $O(\log n)$ fields, where each such field contains a port number. Recall that it is assumed that each port number is encoded using $O(\log n)$ bits. It follows that in the adversary port model, the label size of Scheme π_{rout} is $O(\log^2 n)$.

Let us now consider the designer port model and describe the method by which each vertex u chooses its port numbers, so that the label size of Scheme π_{rout} is $O(\log n \cdot \log\log n)$. Let $E^{sep}(u)$ be the set of edges leading from u to the next vertex on the shortest path connecting u and one of its ancestors in T^{sep}. Since u has $l(u) = O(\log n)$ such ancestors, $E^{sep}(u)$ contains $O(\log n)$ edges. For each edge $e \in E^{sep}(u)$, vertex u chooses a unique port number in the range $\{1, 2, \ldots, l(u)\}$. Therefore, each such port number can be encoded using $O(\log\log n)$ bits. We therefore immediately get that for every vertex v, the port-to-separator sublabel $L^{to-sep}(v)$ can be encoded using $O(\log n \cdot \log\log n)$ bits. We now describe the method by which each vertex u chooses its remaining port numbers, i.e., the port numbers of the edges not in $E^{sep}(u)$. For each such edge e, let $T^i(u)$ be the corresponding subtree formed by u. The corresponding port number at u is set to be the number $l(u) + \rho(T^i(u))$, where $\rho(T^i(u))$ is the number given to $T^i(u)$ by Protocol DYN_SEP. We therefore obtain that the port numbers incident to u are disjoint. As mentioned, for a fixed vertex v and $i \leq l(v)$, the port number $L_i^{from-sep}(v)$ is leading from $s_i(v)$ to x, the next vertex on the shortest path connecting $s_i(v)$ and v. If the edge $(s_i(v), x)$ belongs to $E^{sep}(s_i(v))$ then $L_i^{from-sep}(v)$ is encoded using $O(\log\log n)$ bits. Otherwise, the number of bits in $L_i^{from-sep}(v)$ is $O(\log\log n)$ plus the number of bits used to encode the $i+1$'st subfield in $L^{sep}(v)$. Therefore, the number of bits used to encode $L^{from-sep}(v)$ is at most $O(\log n \cdot \log\log n) + O(\log n) = O(\log n \cdot \log\log n)$. The proof of the following theorem follows.

Theorem 3 *Scheme π_{rout} is a correct dynamic routing scheme that with message complexity $\mathcal{MC}(\pi_{rout}, \bar{n}) = O(n_0 \log^4 n_0) + O(\sum_j \log^4 n_j)$. Moreover, the labels it produces are of optimal length, up to a multiplicative factor of $O(\log\log n)$. I.e., the label size of π_{rout} is $O(\log^2 n)$ for the adversary port model, and $O(\log n \cdot \log\log n)$ for the designer port model.*

7.3 Improved NCA labeling schemes: sketch

In this subsection we sketch the ideas behind our improved dynamic compact NCA labeling scheme. We first describe a static labeling scheme π_{Stat_NCA} supporting the NCA relation on trees. The labels assigned by π_{Stat_NCA} are almost identical to the labels given by the corresponding NCA labeling scheme in [21], with the restriction that in [21] they use the DFS ancestry labeling scheme as a building block to their NCA scheme while we use our π_{Stat_Anc} ancestry scheme instead. (In [21], the id-based NCA problem is referred to as the LCA problem.) Then, using our dynamic ancestry labeling scheme π_{Dyn_Anc} and Protocol HEAVY_CHILD from [19] (which maintains a dynamic δ-heavychild decomposition), we extend the static NCA labeling scheme π_{Stat_NCA} to the dynamic setting.

Let us note that the NCA problem assumes that each vertex has a unique identifier that is encoded using $O(\log n)$ bits, where n is the current number of vertices in the tree. We assume that the identifiers are assigned by an adversary, and that it is maintained that

(1) once an identifier is assigned, it may no longer be changed, and
(2) each identifier is encoded using $O(\log n)$ bits, where n is the current number of vertices in the tree.

Alternatively, we note that our scheme can be slightly modified so that it will also work correctly assuming that the identifiers are chosen by the designer of the algorithm. This can be done using the method in [16], that shows how to assign and maintain disjoint identifiers in the range $[1, 2n]$ at the vertices of the dynamic tree, using message complexity $O(n_0 \log^2 n_0) + O(\sum_j \log^2 n_j)$. One problem that may arise is that in the method of [16], the identifiers are changed occasionally. However, since the changes are made only when the entire tree is rearranged, the NCA scheme may be initialized at that time to overcome this problem. For simplicity of presentation, we do not give a formal description of how to implement this, and assume, instead, that the identifiers are given by an adversary, as discussed above.

For each vertex v, let $id(v)$ denote the identifier of v.

7.3.1 The static NCA labeling schemes π_{Stat_NCA}

In this subsection we describe our static NCA labeling scheme $\pi_{Stat\_NCA} = \langle \mathcal{M}_{Stat\_NCA}, \mathcal{D}_{Stat\_NCA} \rangle$.

We first define the following definitions which are similar to the ones used for the description of the NCA labeling scheme in [21]. For every vertex v and every $0 \leq i \leq depth(v)$, let $\alpha_i(v)$ denote v's ancestor at depth i . In particular, $\alpha_0(v) = r$ and $\alpha_{depth(v)}(v) = v$. Given a δ-heavychild decomposition of T, a non-root vertex is called *small* is the edge leading to its parent in T is light. For every vertex v, the "small ancestor" levels of v are the levels above it in which its ancestor is small,

$$SAL(v) = \{i \mid 1 \leq i \leq depth(v), \alpha_i(v) \text{ is small}\},$$

the *small ancestors* of v are:

$$SA(v) = \{\alpha_i(v) \mid i \in SAL(v)\}.$$

For a vertex v and $1 \leq i \leq depth(v)$, the *i-triple* of v consists of the identifiers of its ancestors on level $i-1$, i and $i+1$,

$$Q_i(v) = \langle (i-1, id(\alpha_{i-1}(v))), (i, id(\alpha_i(v))), \times (i+1, id(\alpha_{i+1}(v))) \rangle.$$

Let $Anc(v)$ be the label assigned to v by our static ancestry labeling scheme π_{Stat_Anc}. For every vertex v we define the *light* sublabel of v to be:

$$L_{light}(v) = \{Q_i(v) \mid 1 \leq i < depth(v), i \in SAL(v)\}.$$

The label $L(v)$ assigned to v by the marker algorithm $\mathcal{M}_{Stat\_NCA}$ is

$$L(v) = \langle id(v), Anc(v), L_{light}(v)\rangle.$$

The decoder $\mathcal{D}_{Stat\_NCA}$ is identical to the decoder $\mathcal{D}_{LCA}$ described in [21] with the restriction that $\mathcal{D}_{Stat\_NCA}$ uses the decoder $\mathcal{D}_{Stat\_Anc}$ of our static ancestry labeling scheme instead of the decoder of the DFS ancestry labeling scheme which is use by the decoder $\mathcal{D}_{LCA}$. By applying Lemma 3.9 and Corollaries 3.12 and 3.14 from [21] to π_{Stat_NCA}, we obtain the following lemma.

Lemma 9 *π_{Stat_NCA} is a correct NCA labeling scheme with label size $O(\log^2 n)$. Moreover, this label size is asymptotically optimal.*

We now describe Protocol $Stat\_NCA$ that is initiated at the root r of a given static tree T and assigns the same labels as π_{Stat_NCA}. Protocol $Stat\_NCA$ uses Sub-protocol LIGHT_NCA(u) which is initiated at the root r of T and assigns the light labels to the vertices of T. Let us first describe Sub-Protocol LIGHT_NCA(u), assuming a δ-heavychild decomposition is given on T.

Sub-protocol LIGHT_NCA(u)

1. If u is a leaf then Sub-protocol LIGHT_NCA(u) terminates.
2. For every child y of u, let $L_{light}(y) = L_{light}(u)$.
3. If u is a non-root vertex and u is a light child of a vertex v then let $i = depth(u)$ and for every child y of u let $L_{light}(y) = L_{light}(y) \circ \langle (i-1, id(v)), (i, id(u)), (i+1, id(y))\rangle$.
4. Each child y of u, invokes Sub-protocol LIGHT_NCA(y).

Clearly, sub-protocol LIGHT_NCA can be implemented using $O(n)$ messages of size $O(\log^2 n)$. Since we restricted the message size to be $O(\log n)$, we obtain the following.

Lemma 10 $\mathcal{MC}(\text{LIGHT\_NCA}, n) = O(n \log n)$.

Protocol Stat_NCA

1. The root broadcasts a signal instructing all the tree vertices to calculate their exact weights and depths through a convergecast process. During the convergecast process, each vertex v keeps a pointer to its heaviest child $h(v)$, i.e., a child u of v such that no other child of v has more descendants than u.

   * A 1/2-heavychild decomposition is calculated. *
2. Invoke Protocol STAT_ANCEST and for every vertex v, let $Anc(v)$ be the label assigned to v by Protocol STAT_ANCEST.
3. Invoke Protocol LIGHT_NCA(r) after which every vertex v is assigned the light sublabel $L_{light}(v)$.
4. For every vertex v, let $L(v) = \langle id(v), Anc(v), L_{light}(v)\rangle$.

By Lemmas 9 and 10, Theorem 2 and the fact that Protocol $Stat\_NCA$ assigns the same labels as π_{Stat_NCA}, we obtain the following lemma.

Lemma 11 *π_{Stat_NCA} is a correct NCA labeling scheme with the following complexities.*

1. $\mathcal{LS}(Stat\_NCA, n) = \Theta(\log^2 n)$,
2. $\mathcal{MC}(Stat\_NCA, n) = O(n \log n)$.

7.3.2 The dynamic NCA labeling scheme π_{Dyn_NCA}

In this subsection we present our dynamic NCA labeling scheme $\pi_{Dyn\_NCA} = \langle \mathcal{M}_{Dyn\_NCA}, \mathcal{D}_{Dyn\_NCA}\rangle$. For simplicity of presentation, we describe the scheme assuming the serialized model. We note however, that one can rather easily adapt the scheme for the controlled model.

The decoder $\mathcal{D}_{Dyn\_NCA}$ is the same as the decoder of the corresponding static scheme $\mathcal{D}_{Stat\_NCA}$. Protocol Dyn_NCA first runs Protocol Dyn_Anc that maintains the ancestry label $Anc(v)$ at each vertex v during the dynamic scenario. In addition, Protocol Dyn_NCA runs Protocol HEAVY_CHILD from [19] in order to maintain a δ-heavychild decomposition of the dynamic tree. We note that Protocol HEAVY_CHILD from [19] was designed for the leaf-increasing tree model, however, one can obtain a similar protocol in the leaf-dynamic setting, using the method of [16] (instead of the method of [3] that the protocol of [19] is based upon). The resulted Protocol HEAVY_CHILD has message complexity $O(n_0 \log^2 n_0) + O(\sum_i \log^2 n_i)$.

Protocol Dyn_NCA

1. Invoke Protocol DYN_ANCEST and for every vertex v, let $Anc(v)$ be the label assigned by Protocol DYN_ANCEST to vertex v.
2. Invoke Protocol HEAVY_CHILD.
3. Whenever a vertex v of depth i changes its pointer $h(v)$ from u^{old} to u^{new} in Protocol HEAVY_CHILD then v initiates the following.
   (a) Initialize $L_{light}(u^{old}) = L_{light}(u^{new}) = L_{light}(v)$
   (b) Invoke sub-protocols LIGHT_NCA(u^{old}) and LIGHT_NCA(u^{new}).
4. For every vertex v, let $L(v) = \langle id(v), Anc(v), L_{light}(v)\rangle$.

Theorem 4 *π_{Dyn_NCA} is a correct dynamic labeling scheme supporting the NCA relation and has the following complexities.*

1. $\mathcal{LS}(Dyn\_NCA, n) = \Theta(\log^2 n)$,
2. $\mathcal{MC}(Dyn\_NCA, \bar{n}) = O(n_0 \log^4 n_0) + O(\sum_i \log^4 n_i)$.

Proof The correctness of π_{Dyn_NCA} follows from the correctness of π_{Dyn_Anc} and from the fact that at any quiet time, given the current heavychild decomposition maintained by Protocol HEAVY_CHILD, the labels assigned by Protocol Dyn_NCA are the same as the labels assigned by $Stat\_NCA$ for the same heavychild decomposition. The fact that $\mathcal{LS}(Dyn\_NCA, n)) = \Theta(\log^2 n)$ follows from Theorem 2, Lemma 11 and from Claim 3.5 in [19] where they prove that Protocol HEAVY_CHILD maintains a 3/4-heavychild decomposition.

Let us now show why $\mathcal{MC}(Dyn\_NCA, \bar{n}) = O(n_0 \log^4 n_0) + O(\sum_i \log^4 n_i)$. The fact that Steps 1 and 2 in Protocol Dyn_NCA incur $O(n_0 \log^4 n_0) + O(\sum_i \log^4 n_i)$ messages follows from Theorem 2 and the fact that Protocol HEAVY_CHILD incurs $O(n_0 \log^2 n_0) + O(\sum_i \log^2 n_i)$ messages. The proof that Steps 3(a) and 3(b) in Protocol Dyn_NCA incur $O(n_0 \log n_0) + O(\sum_i \log n_i)$ messages throughout the dynamic scenario follows the same steps as the proof of Lemma 3.6 in [19]. The theorem follows. □

7.4 Extended distance labeling schemes on dynamic trees: sketch

In [17], they assume the serialized dynamic model and give two β-approximate distance labeling schemes on dynamic trees assuming that the vertices are fixed but the edge weights may change (as long as they remain positive). Informally, both schemes are based on the following principle. In a preprocessing stage, a (static) separator decomposition is calculated on the tree. In this decomposition, each vertex belongs to $O(\log n)$ subtrees, one for each level of the recursion. A mechanism for estimating the distance to the root is applied separately to each of the decomposed subtrees. Therefore, each vertex v participates in $O(\log n)$ such protocols, each corresponding to a subtree that v belongs to. This enables v to maintain estimates to the distances between v and the roots of these subtrees. These distance estimates are then encoded in v's label. The distance between any two vertices in the dynamic tree can be retrieved from their corresponding lists of estimates, which are encoded in their labels.

Using our dynamic separator decomposition π_{Dyn_Sep} for the leaf-dynamic model, and applying the above mentioned principle on the resulted dynamic separator decomposition, the schemes in [17] can be modified to operate correctly under more general dynamic models, allowing also leaves to be either added or removed from the tree. Moreover, the extended dynamic schemes incur only an extra additive $O(n_0 \log^4 n_0) + O(\sum_j \log^4 n_j)$ factor to the message complexity of the original schemes.

8 Conclusion

Our improved ancestry, routing and NCA labeling schemes apply for the controlled model, which is less restricted than the serialized model, for which the previous schemes where given. Moreover, in the leaf-increasing model, our dynamic schemes can operate under the weak *uncontrolled* model in which the topological changes may occur in rapid succession or even concurrently (correctness, however, is still guaranteed only at quiet times). This can be achieved by simulating the controllers assuming the controlled model, and ignoring new vertices which have not received the proper permits for entering the tree. It may be useful to design such schemes for even weaker dynamic models, especially ones which are more robust under faults (for example, using backup procedures). In addition, it would be interesting to design dynamic schemes which can operate under more types of topology changes, for example, ones which can operate also under additions and deletions of internal vertices. In does not seem likely that a dynamic separator would operate efficiently under such topology changes (for example, if the level 1 separator is removed, all vertices should be notified). However, it does seem reasonable that one could find an efficient ancestry scheme under such topology changes, especially since the controller of [16] can operate under such topology changes.

The main factor influencing the message complexity of our schemes is the message complexity of the $(M, M/2)$-Controllers of [16] which is $O(n_0 \log^2 n_0) + O(\sum_j \log^2 n_j)$. A slight improvement in the message complexity of such controller (if possible) would immediately improve the message complexity of all our schemes.

Regarding the label size, we were able to construct dynamic labeling schemes on trees, with polylogarithmic amortized message complexity and optimal label size for the ancestry and NCA relations. However, it still remains to show whether one can construct compact routing labeling schemes on trees, with polylogarithmic amortized message complexity. In addition, this paper does not consider labeling schemes supporting the label-based NCA relation on trees. Constructing an efficient compact scheme for that function seems to be a challenging task. We leave these questions open.

References

1. Abiteboul, S., Alstrup, S., Kaplan, H., Milo, T., Rauhe, T.: Compact labeling scheme for ancestor queries. SIAM J. Comput. **35**(6), 1295–1309 (2006)

2. Abiteboul, S., Kaplan, H., Milo, T.: Compact labeling schemes for ancestor queries. In: Proc. 12th ACM-SIAM Symp. on Discrete Algorithms, January (2001)
3. Afek, Y., Awerbuch, B., Plotkin, S.A., Saks, M.: Local management of a global resource in a communication. J. ACM **43**, 1–19 (1996)
4. Afek, Y., Gafni, E., Ricklin, M.: Upper and lower bounds for routing schemes in dynamic networks. In: Proc. 30th Symp. on Foundations of Computer Science, pp. 370–375 (1989)
5. Alstrup, S., Gavoille, C., Kaplan, H., Rauhe, T.: Nearest common ancestors: a survey and a new distributed algorithm. Theory Comput. Syst. **37**, 441–456 (2004)
6. Alstrup, S., Rauhe, T.: Small induced-universal graphs and compact implicit graph representations. In: Proc. 43rd IEEE Symp. on Foundations of Computer Science, November (2002)
7. Cole, R., Hariharan, R.: Dynamic LCA queries on trees. SIAM J. Comput. **34**(4), 894–923 (2005)
8. Eppstein, D., Galil Z., Italiano, G.F.: Dynamic graph algorithms. In: M.J. Atallah, (ed.) Algorithms and Theoretical Computing Handbook, Chap. 8. CRC Press, Boca Raton (1999)
9. Fraigniaud, P., Gavoille, C.: Routing in trees. In: Proc. 28th Int. Colloq. on Automata, Languages & Prog., LNCS, vol. 2076, pp. 757–772, July (2001)
10. Fraigniaud, P., Gavoille, C.: A space lower bound for routing in trees. In: Proc. 19th Symp. on Theoretical Aspects of Computer Science, pp. 65–75, March (2002)
11. Feigenbaum, J., Kannan, S.: Dynamic graph algorithms. In: Handbook of Discrete and Combinatorial Mathematics. CRC Press, Boca Raton (2000)
12. Gavoille, C., Katz, M., Katz, N.A., Paul, C., Peleg, D.: Approximate distance labeling schemes. In: 9th European Symp. on Algorithms, pp. 476–488, August (2001)
13. Kannan, S., Naor, M., Rudich, S.: Implicit Representation of Graphs. SIAM J. Discrete Math. **5**, 596–603 (1992)
14. Korman, A.: General Compact Labeling schemes for dynamic trees. In Proc. 19th Symp. on Distributed Computing, September (2005)
15. Korman, A.: Labeling Schemes for vertex connectivity. In: Proc. 34th Int. Colloq. on Automata, Languages and Prog., July (2007)
16. Korman, A. Kutten, S.: Controller and estimator for dynamic networks. In: Proc. 26th ACM Symp. on Principles of Distributed Computing, August (2007)
17. Korman, A., Peleg, D.: Labeling schemes for weighted dynamic trees. In: Proc. 30th Int. Colloq. on Automata, Languages & Prog., July (2003)
18. Korman, A., Peleg, D.: Dynamic routing schemes for general graphs. In: Proc. 33rd Int. Colloq. on Automata, Languages & Prog. (2006)
19. Korman, A., Peleg, D., Rodeh, Y.: Labeling schemes for dynamic tree networks. Theory of Computing Systems 37(1), Special Issue of STACS'02 papers, pp. 49–75 (2004)
20. Peleg, D.: Distributed Computing: a :ocality-sensitive Approach. SIAM, Philadelphia (2000)
21. Peleg, D.: Informative labeling schemes for graphs. Theoretical Computer Science **340**, Special Issue of MFCS'00 papers, pp. 577–593 (2005)
22. Peterson, L.L., Davie, B.S.: Computer Networks: A Systems Approach. Morgan Kaufmann, San Francisco (2007)
23. Schieber, B., Vishkin, U.: On finding lowest common ancestors: simplification and parallelization. SIAM J. Comput. **17**(6), 1253–1262 (1988)
24. Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. J. Comput. Syst. Sci. **26**(1), 362–391 (1983)
25. Tanenbaum, A.S.: Computer Networks. Prentice Hall, Englewood Cliffs (2003)
26. Thorup, M., Zwick, U.: Compact routing schemes. In: Proc. 13th ACM Symp. on Parallel Algorithms and Architecture, pp. 1–10, July (2001)