Improved Compact Routing Schemes for Dynamic Trees

[Extended Abstract]

Amos Korman *

ABSTRACT

A classical routing problem consists of assigning a label and distinct port numbers to each node of a graph, such that for every node v, given its own label and the label of any destination vertex u, node v can find which of its incident port numbers leads to the next vertex on a shortest path connecting v and u. In the static (fixed topology) setting, such a routing scheme is evaluated by the *label size*, i.e., the maximal number of bits stored in a label. Naturally, special attention is given to *compact* schemes, which are schemes enjoying asymptotically optimal labels.

Many routing schemes were proposed for the static setting. However, the more realistic and complex dynamic setting, in which topology changes may occur at arbitrary nodes, has received much less attention. In the dynamic setting, the occurrence of topology changes may force the scheme to occasionally update the (hopefully short) labels, by delivering information from place to place. This raises a natural tradeoff between the size of the labels and the number of messages required for maintaining them.

The above *dynamic routing* problem was proposed by Afek, Gafni, and Ricklin (1989), who also presented an elegant and rather efficient dynamic routing scheme for trees, supporting one type of topology change, namely, the addition of a leaf. Various attempts for improving the tradeoff between the label size and the message complexity as well as for supporting more types of topology changes on trees, were subsequently proposed. Still, the best known compact routing scheme for dynamic trees has very high message complexity, namely, $O(n^{\epsilon})$ amortized messages per topological change. Moreover, previous routing schemes for dynamic trees support at most two kinds of topology changes, namely, the addition and the removal of a leaf node.

In this paper, we present two compact routing schemes for dynamic trees that incur extremely low message complexity and can support more types of topology changes than previous schemes. We first present a dynamic compact routing scheme that supports the additions of both leaves and internal nodes and incurs only $O(\log n)$ amortized message complexity per node. We then extend that scheme obtaining a dynamic compact routing scheme that supports additions of both leaves and internal nodes as well as deletions of nodes of degree at most 2. The extended scheme incurs $O(\log^2 n)$ amortized message complexity per topological change.

*CNRS and Université Paris Diderot - Paris 7, France. E-mail: amos.korman@gmail.com. Supported in part by the ANR project ALADDIN, by the INRIA project GANG, and by COST Action 295 DYNAMO.

PODC'08, August 18–21, 2008, Toronto, Ontario, Canada.

Categories and Subject Descriptors

C.2.2 [**Network Protocols**]: [Routing protocols]; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Network problems, Graph labeling*; E.1 [**Data structures**]: [Distributed data structures, Trees]

General Terms

Algorithms, Theory

Keywords

Distributed algorithms, Dynamic networks, Routing schemes, Ancestry labeling schemes, Dynamic name assignment.

1. INTRODUCTION

Background. Network representations play an important role in many domains of computer science, ranging from data structures and graph algorithms to distributed computing and communication networks. The study of methods for representing a network, and specifically, for designing a routing scheme, is rather well developed in the static (fixed topology) setting. Typically, the main goal of such a scheme is to minimize the *label size*, i.e., the maximal local information (label) stored at a node (see e.g., [2, 4, 10, 11]). Not surprisingly, the main focus is on *compact* schemes, which are schemes with asymptotically optimal label size.

Though many representation schemes were given for the static setting, the more complex *dynamic* setting, in which processors may join or leave the network or new connections may be established or removed, has received much less attention in the literature. In the dynamic scenario, in order to maintain the labels, the scheme needs to occasionally update labels following a topology change, which may require the delivery of information from place to place. This raises the problem of maintaining short labels (hopefully asymptotically optimal) using as few as possible messages.

The current paper investigates the problem of maintaining a routing scheme in a dynamic tree. The problem was originally introduced by [3], and later studied in a series of papers [14, 16, 17], which proposed different tradeoffs between the label size and the message complexity. Still, the best known compact routing scheme on dynamic trees [14] has very high message complexity, namely, $O(n^\epsilon)$ amortized messages per topological change. Moreover, the types of topology changes supported by previous schemes on trees are limited to additions and removals of leaves. On top of that, the correctness of previous schemes is guaranteed only for *quiet* times, in which all updates concerning the previous topology changes, have already occurred.

The main contribution of this paper is the construction of two compact routing schemes for dynamic trees that incur extremely low message complexity. Moreover, our schemes can support more types of topology changes than previous schemes and are guaranteed to be correct at all times. Specifically, we first present a dynamic compact routing scheme that supports the additions of both leaves and internal nodes, and uses only logarithmic amortized message complexity per node. We then extend that scheme obtaining a dynamic compact routing scheme that supports additions of both leaves and internal nodes as well as removals of nodes of degree at most 2. The extended scheme enjoys $O(\log^2 n)$ amortized message complexity[1].

Related work. The concept of implicit labeling schemes was introduced by [12], which also gave an elegant and simple static *ancestry labeling scheme* using label size $\Theta(\log n)$. An ancestry labeling scheme assigns labels to the nodes of a tree, so that ancestry queries between two nodes can be answered simply by looking at the corresponding labels. Besides being a known tool for constructing routing schemes on trees, ancestry labeling schemes were also found useful for several applications to XML search engines. This motivated some very interesting attempts for improving the constant factor in the label size [4, 13].

Routing schemes on static trees were investigated in a number of papers e.g., [20, 22]. Compact routing labeling schemes on static trees were given independently in [10, 21]. Specifically, for the *designer port* model (considered also in this paper), in which the designer of the scheme can freely enumerate the port numbers of the nodes, [10] and [21] showed how to construct a routing scheme using labels of size $\Theta(\log n)$.

Dynamic routing and ancestry labeling schemes on trees where studied in [3, 14, 16, 17]. In [3] the authors gave a dynamic routing scheme on dynamically growing trees (in which only leaves are allowed to join) using labels of size $O(\Delta \log^3 n)$ (where Δ is the maximum degree in the tree) and amortized message complexity $O(\log n)$ per node, where each message is encoded using $O(\log^2 n)$ bits. (In this paper, we assume that each message is encoded using $O(\log n)$ bits.) Again, if only leaves can join the tree, [14, 17] gave a dynamic routing scheme with label size $O(\log^2 n)$ and amortized message complexity $O(\log n)$. In the case where leaves can also be removed from the network, [16] gave a dynamic routing scheme with amortized message complexity $O(\log^4 n)$ and label size that is larger than the optimum by a factor $O(\log \log n)$. The best known compact routing scheme on trees [14] uses $O(n^\epsilon)$ amortized message complexity, where $0 < \epsilon < 1$ is some constant. Previous routing schemes on dynamic trees support merely additions and removals of leaves, and are guaranteed to be correct only at quiet times.

The best known compact ancestry scheme on dynamic trees [16] uses $O(\log^4 n)$ amortized message complexity. That scheme supports additions and removals of leaves and can be extended using the method of [15] to support also removals of internal nodes. Again, its correctness is guaranteed only for quiet times.

In [1] and [15], the authors study the controller problem on dynamic trees; in [1] they assumed that only leaves can join the tree, and in [15] they considered the case where both leaves and internal nodes can either be added to or be removed from the tree. The controllers presented in the above papers can be used for several applications. In particular, they can be used to solve the *dynamic name assignment* problem, by maintaining disjoint $\log n + O(1)$-bit identities (or labels) at the nodes of the dynamic tree, using $O(\log^2 n)$ amortized message complexity. The same message complexity can be used to maintain at each node, a constant approximation to the number of nodes in the tree. As shown in [15] (and used also here), their approximation scheme can be used to extend several types of labeling schemes to support also removals of nodes. The extension is made with an extra additive cost of $O(\log^2 n)$ to the amortized message complexity.

In general, the design of distributed protocols for problems in dynamic trees has been an important theoretical issue in distributed computing [1, 3, 5, 7, 9, 15, 18].

Contribution and outline. In Section 3 we first construct our compact ancestry labeling scheme for growing trees (both leaves and internal nodes are allowed to join the tree) which incurs $O(\log n)$ amortized message complexity, per node.

Any dynamic ancestry labeling scheme also solves the *dynamic name assignment* problem (see [1, 15]) by maintaining disjoint identities (labels) at the nodes of the changing tree. Our dynamic ancestry scheme solves this problem on a growing tree, using a different tradeoff than the one in [1, 15], namely, we maintain $O(\log n)$ bit identities using only $O(\log n)$ amortized message complexity.

Using the method of [15], we extend our ancestry scheme to support not only insertions but also deletions of leaves and internal nodes. The extended scheme incurs $O(\log^2 n)$ amortized message complexity.

Due to lack of space, the detailed description and analysis of our dynamic ancestry labeling schemes are deferred to the full paper.

[1]In fact, the message complexity is given by a somewhat more complicated formula, namely, $O(n_0 \log^2 n_0) + O(\sum_i \log^2 n_i)$, where n_0 is the initial number of nodes in the tree and n_i is the number of nodes when the i'th topology change takes place. Note, if the number of topology changes is $\Omega(n_0)$ (in particular, if the initial tree contains just the root), then the message complexity is simply $O(\sum_i \log^2 n_i)$. Also, if there are no removals of nodes, then the message complexity is $O(n \log^2 n)$, where n is the final number of nodes. For simplicity, as in [14, 16, 17], we abuse the term $O(\log^j n)$ *amortized message complexity* to denote the message complexity $O(n_0 \log^j n_0) + O(\sum_i \log^j n_i)$.

Our dynamic ancestry scheme is then used in Section 4 to construct our dynamic routing scheme for a growing tree, which assigns and maintains asymptotically optimal $\Theta(\log n)$-bit labels, and incurs $O(\log n)$ amortized message complexity, per node. Informally, with our dynamic ancestry labeling scheme, a node that wishes to deliver a message to some other node, knows whether it should forward the message to its parent or to one of its children. However, if it needs to deliver the message to one of its children, it still needs to know to which child to deliver the message. This problem is then tackled by adapting the principles of the static routing scheme in [10] to the more complex dynamic setting.

Our routing scheme is then extended to support also removals of of degree 1 and 2 vertices using the method mentioned in [15]. Similarly to ancestry case, our extended dynamic routing scheme also incurs $O(\log^2 n)$ amortized message complexity.

In contrast to the previous ancestry and routing schemes on dynamic trees, whose correctness is guaranteed merely at quiet times, our schemes are correct at all times. Table 1 summarizes the complexities of our dynamic routing schemes in comparison to previously known results.

We would like to point out that our schemes as well as the dynamic schemes in [14, 17, 16], assume the *controlled* dynamic model (see Section 2), in which the topology changes do not occur spontaneously, and instead can be delayed by the update protocol. Clearly, no scheme can be expected to be both compact and correct at all times, if vertices are being inserted to the tree in a very rapid succession. In the case where only leaves may join the tree (as in [3]), our schemes can be extended to support spontaneous insertions of leaves, however, in this case, the correctness is only guaranteed at quiet times (similarly to [3]).

2. PRELIMINARIES

We consider the standard asynchronous point-to-point message passing model. The network topology is described by an undirected communication tree $T = \langle V, E\rangle$, where V is a set of vertices, representing processors, and E is a set of edges, representing bidirectional communication links. The communication between vertices is made by exchanging messages over the edges. Each message is encoded using $O(\log n)$ bits, where n is the current tree size. A message sent over an edge arrives without errors at the other end, at an arbitrary but finite delay. We assume that the computations performed at a node are made instantaneously. To reduce issues related to concurrency within one node, only one message is received by a node at a time.

The ancestry relation in the tree T is defined as the transitive closure of the parenthood relation (in particular, a vertex is an ancestor of itself). For every vertex $v \in T$, the *depth* of v, is the (unweighted) distance between v and the root r. For a vertex v, let T_v denote the subtree of T hanging down from v (including v), and let $\omega(v)$, denote the number of vertices in T_v, i.e., $\omega(v) = |T_v|$. We refer to $\omega(v)$ as the *weight* of v.

Static labeling schemes on trees. Let f be a function on pairs of vertices in a tree T. In the routing case, given two vertices w and v, the routing function $f(w, v)$ is defined as the port number at w leading to the next vertex on the shortest path connecting w and v. In the ancestry case, the ancestry function $f(w, v)$ equals 1, if w is an ancestor of v, and 0, otherwise.

A static *f-labeling scheme* on trees is composed of the following components:
1) A *marker* algorithm that given a tree T assigns a label $\mathrm{L}(v)$ to each vertex $v \in T$.
2) A *decoder* algorithm that given the labels $\mathrm{L}(w)$ and $\mathrm{L}(v)$ of two vertices w and v in T, outputs $f(w, v)$. (The decoder, in the routing case, is also referred to as a *router*.)

The most common measure to evaluate a static labeling scheme is the *label size*, i.e, the maximum number of bits used in a label. A labeling scheme with asymptotically optimal label size is called *compact*.

The dynamic models. At any given time, let n denote the current tree size. We assume that we may initialize the labels and data structure of the initial tree, in a *preprocessing stage*, which is performed before the dynamic scenario starts. (Clearly, no scheme can be expected to be correct before the nodes are assigned labels.)

We consider the following types of topology changes. **(1)** *Add-leaf:* A new degree one vertex u is added as a child of an existing vertex v. **(2)** *Add internal vertex (between neighbors v and w):* Edge $e = (v, w)$ splits into two edges (v, u) and (u, v) for a new vertex u. If v was w's parent, then u is a child of v and w is considered a child of u. **(3)** *Remove-leaf:* A (non-root) vertex v of degree one is deleted. **(4)** *Remove internal vertex:* A vertex u whose degree in T is larger then one is deleted. The vertices which were u's children in the tree, become the children of u's parent. It is assumed that the root is never removed.

In the *growing tree* model it is assumed that merely the first two types of topology changes may occur, namely, Add-leaf and Add internal vertex. A *growing tree* consists of an initial tree that may change according to the growing tree model, and a *growing path* is a growing tree which is initially a path, and remains one throughout the dynamic scenario, i.e., the path may undergo insertions of internal nodes and insertions of a leaf, but only as a child of the bottom most node.

In the dynamic setting, the definition of a decoder (or router) of an *f-labeling scheme* remains the same, however, the marker protocol changes to a distributed *update protocol*, whose goal is to assign and maintain the labels to fit the requirements of the decoder (or router).

We consider the *controlled* dynamic model (considered also in [1, 9, 15, 16, 19]), in which the topological changes do not occur spontaneously. Instead, when an entity wishes to cause a topological change at some vertex u, it enters a *request* at u. This request triggers the invocation of the update protocol at u, which must grant a *permit* to the request after a finite time. For this purpose, and for the purpose of updating the labels, the update protocol may send messages over the links of the underlying tree. The requested topology change is performed only after the request is granted a permit from the update protocol. This model can be found useful in various contexts of overlay networks applications, where many of the topology changes are ones that are decided by the designer of the algorithm, and therefore can be delayed, possibly beyond their inherent delay. (For more details regarding the applications of the controlled model, see [15].)

A request to insert or remove a vertex u arrives at its parent. As mentioned, once a request is granted a per-

Routing schemes	Paper [3]	Paper [14]	Paper [16]		This paper (result 1)	This paper (result 2)
Label size	$O(\Delta \log^3 n)$	$\Theta(\log n)$	$O(\log n \cdot \log\log n)$		$\Theta(\log n)$	$\Theta(\log n)$
Amortized message complexity	$O(\log n)$	$O(n^\epsilon)$	$O(\log^4 n)$		$O(\log n)$	$O(\log^2 n)$
Types of topological changes	add leaf	add leaf remove leaf	add leaf remove leaf		add any node	add any node remove a leaf or deg 2 node
Correctness guarantee	at quiet times	at quiet times	at quiet times		at all times	at all times

Table 1: The table summarizes the performances of our dynamic routing schemes in comparison previous ones.

mit, the corresponding topology change may occur. Performing the actual topology change may require the use of some additional protocols, which may not be the same in different types of networks. For example, in the case of removing some vertex u, the requesting entity may need to perform a handshake with the neighboring nodes to verify that no messages are on their way to u. Alternatively, a loss can be allowed temporarily, and some form of acknowledgement and retransmission may be used to recover from that loss. Such additional protocols may be interesting themselves. However, they are beyond the scope of the current paper. Instead, for simplicity, we assume that once a request is granted a permit, the corresponding topology change occurs instantaneously. In addition, if a vertex u is inserted, then it is inserted together with a label assigned to it by the vertex inserting it, i.e., u's parent. See [15] for more details regarding the dynamic model.

When a new edge is attached to a vertex v, the corresponding port at v is assigned a unique port-number, i.e., at any given moment, the port numbers at v are distinct. We consider the *designer port* model which allows each vertex v to freely select and change the port numbers on its incident ports.

3. THE DYNAMIC ANCESTRY SCHEME

In this section we first consider the growing tree model, and construct a dynamic compact ancestry labeling scheme `Dyn-Anc`, that enjoys $O(n \log n)$ message complexity (i.e., $O(\log n)$ messages per node). We then extend that scheme to support also deletions of nodes, at the cost of increasing the amortized message complexity from $O(\log n)$ to $O(\log^2 n)$. Due to lack of space, this extended abstract contains just a very brief overview; the detailed description and analysis of the protocols are deferred to the full paper.

In [9], the authors consider the problem of counting the number of nodes (or events) in dynamic trees, in the synchronous setting. As a first step, they give a novel method for reducing the problem to the restricted case in which the underlying dynamic tree is a dynamic path. We first claim that a similar reduction can be used to reduce the problem of constructing an ancestry scheme on a dynamic tree to the problem on a dynamic path (even for the asynchronous setting).

We next provide a protocol that implements a compact ancestry labeling scheme on a growing path. This protocol can be considered as a (simplified) deterministic and asynchronous variant of the main protocol in [9] (or as a distributed asynchronous implementation of the main sequential algorithm in [6]).

THEOREM 3.1. *There exists a dynamic compact ancestry labeling scheme on a growing tree, which incurs $O(n \log n)$ message complexity.*

Since any ancestry labeling scheme maintains disjoint identifiers, our dynamic scheme is in fact also a dynamic name assignment protocol, that achieves a different tradeoff (between the identity size and the message complexity) from the protocols in [1, 15].

COROLLARY 3.2. *There exists a dynamic name assignment protocol on a growing tree that assigns and maintains $O(\log n)$-bit disjoint identities, using $O(n \log n)$ messages.*

We then extend our dynamic ancestry scheme `Dyn-Anc` on growing trees to support also deletions of nodes. The idea behind the extension is similar to the one presented in [15]. Informally, by simulating the behavior of deleted nodes by existing ones, we can implement an ancestry labeling on a dynamic tree, supporting insertions and removals of both leaves and internal nodes. However, the label size of such a scheme is asymptotically optimal with respect to N, where N is the total number of vertices existing in the scenario, including deleted vertices. Therefore, if the tree size falls significantly, then the current labels which are small with respect to N, might become too large with respect to the current number of nodes n in the tree. To tackle this issue, we run, in parallel to Protocol `Dyn-Anc`, the protocol from [15], for estimating the number of topological changes in the tree. Every $\Theta(n)$ topological changes, we initialize the labels and data structures of the vertices by running Protocol `Dyn-Anc` from scratch. Since this size estimation protocol incurs $O(n_0 \log^2 n_0) + O(\sum_i \log^2 n_i)$ message complexity (i.e., $O(\log^2 n)$ amortized message complexity, per topology change), we obtain the following theorem.

THEOREM 3.3. *There exists a dynamic compact ancestry labeling scheme on trees, supporting insertions and removals of both leaves and internal vertices, using message complexity $O(n_0 \log^2 n_0) + O(\sum_i \log^2 n_i)$, where n_i is the number of nodes in the tree, after the i'th topological change.*

4. THE DYNAMIC ROUTING SCHEME

In this section we first consider the growing tree model and establish our compact routing scheme `Dyn-Rout` for a growing tree, which incurs $O(n \log n)$ message complexity. We then extend that scheme to support also deletions of nodes of degree at most 2. The extended scheme incurs $O(\log^2 n)$ amortized message complexity.

4.1 The dynamic routing scheme in the growing tree model

Let us denote the dynamic compact ancestry scheme on trees given by Theorem 3.1, by Protocol `Dyn-Anc`. Scheme `Dyn-Rout` runs Protocol `Dyn-Anc` which assigns each vertex v an *ancestry label* $\mathrm{L}_{anc}(v)$. Given its own ancestry label and the ancestry label of a destination vertex u, vertex v can find whether u is its descendant or not. Note that if u is not a descendant of v then the neighbor of v on the shortest path connecting v and u is v's parent $p(v)$. We let the port number leading from each non-root vertex to its parent be zero, at all times. Thus, if u is not a descendant of v then the router at v simply outputs 0. However, if u is a descendant of v then the message should be delivered to one of v's children, the one which is a ancestor of u. The rest of the section is dedicated to show how short labels and disjoint port numbers can be efficiently maintained so that, in the above case, given the labels of v and u, vertex v will know how to find the port number leading to its child which is an ancestor of u.

Our general strategy is to adapt the principles of the static scheme in [10] into the dynamic scenario. In addition to other components, which will be described later, the label $\mathrm{L}(v)$ of a vertex v contains also the following three sublabels: the *ancestry* label $\mathrm{L}_{anc}(v)$, the *heavy* label $\mathrm{L}_{heavy}(v)$ and the *light* label $\mathrm{L}_{light}(v)$. It is maintained that at all times, one of the port numbers at each non-leaf vertex v is set to be 1. The corresponding child of v is called *heavy* and is denoted by $\mathrm{H}(v)$; All other children of v are called *light*. The *light subtree* of v, denoted $T_{light}(v)$, is the subtree containing v, all its light children and all their descendants. The *light weight* of v, denoted $\omega_{light}(v)$, is the number of nodes in $T_{light}(v)$, i.e., $\omega_{light}(v) = |T_{light}(v)|$. (Note that the light weight of v equals to its weight minus the weight of his heavy child.)

The role of the heavy label at v is to allow v to know whether the destination vertex u is a descendant of $\mathrm{H}(v)$ or not. More precisely, we would like to be able to determine whether u is a descendant of $\mathrm{H}(v)$ at a given time, simply by looking at the labels $\mathrm{L}_{heavy}(v)$ and $\mathrm{L}_{anc}(u)$, at the given time. This, of course, can be done if $\mathrm{L}_{heavy}(v)$ would be the same as $\mathrm{L}_{anc}(\mathrm{H}(v))$, at all times. However, maintaining this equality at all times is impossible due to the asynchronous nature of the setting. Still, one can allow v to know (at all times) whether it's descendant u is a descendant of $\mathrm{H}(v)$, using a rather simple handshake procedure, that is invoked between $\mathrm{H}(v)$ and v, whenever the ancestry label of $\mathrm{H}(v)$ is updated. If v finds that the destination vertex u is a descendant of its heavy child $\mathrm{H}(v)$, the router at v simply outputs 1 (the port number leading to $\mathrm{H}(v)$).

The role of the light labels is to allow v to determine (in the case where its descendant u is not a descendant of $\mathrm{H}(v)$) which of its light children w is an ancestor of u. The static scheme of [10] proposed the following. The light label at $\mathrm{H}(v)$ is the same as the light label at v. In contrast, assuming (distinct) port numbers are assigned to the ports at v, the light label at each light child w of v is the light label at v concatenated with $Port(v, w)$, the port number at v leading to w, i.e., $\mathrm{L}_{light}(w) = L_{light}(v) \circ Port(v, w)$. This recursive definition enables v to extract the port number leading to its light child w which is an ancestor of u, simply by comparing the light labels at v and u. We refer to the procedure that compares the light labels and extracts the port number $Port(v, w)$ as Procedure Extract.

Note that since each light label is a concatenation of port numbers, the size of a light label is determined by the sum of the sizes of the concatenated port numbers, which still need to be assigned. The main trick in [10] was to choose the heavy child $\mathrm{H}(v)$ to be a child of v with maximal weight, and to assign the port numbers at v leading to its light children, in a way, such that higher port numbers are assigned to children with fewer descendants. The choice of the heavy children, guarantees that the total number of port numbers in a light label is $O(\log n)$, and the choice of the port numbers leading to light children, guarantees that a light label does not consists of many large port numbers. By combining these two restrictions on the port numbers in a light label, the $O(\log n)$ label size follows.

We would have liked to adapt the above trick to the dynamic setting. However, we could not find an efficient way for maintaining it at all times (or even just at quiet times). In particular, it seems that merely ensuring that the weight of the heavy child of v is relatively large with respect to the weights of the other children of v, would require the scheme to maintain the approximate weight of $\mathrm{H}(v)$, and the best known scheme that does that [1, 15] already uses $O(n \log^2 n)$ messages. Instead of applying the above trick at all times, we apply it only occasionally, whenever we invoke Procedure `Shuffle`, that is used for balancing and reorganizing certain portions of the tree.

Before dwelling into the description of Procedure `Shuffle`, let us first describe the update protocol, which is responsible for scheduling and triggering the different applications of Procedure `Shuffle`.

The update protocol. A vertex may be either *frozen* or *unfrozen*. Initially, all vertices are unfrozen. When a request (for inserting a child) arrives at an unfrozen vertex, it is handled by Protocol `Dyn-Anc`, that eventually grants it a permit. In contrast, when a request arrives at a frozen vertex v, it is first put in a queue at v. When v becomes unfrozen again (which is guaranteed to occur eventually), the requests from its queue are dequeued one by one, according to the First In First Out discipline, and are handled by Protocol `Dyn-Anc`. If Protocol `Dyn-Anc` issues a permit to a request at a frozen vertex v, then v delays the performance of the corresponding topology change until it becomes unfrozen again. In contrast, if v is unfrozen when the permit arrives, then the following happen. If the request is for inserting a leaf w then v first simulates Procedure `Shuffle` on the subtree containing itself and its future child w (this simulation takes zero time). It then inserts w with the appropriate light and heavy labels which are given to it by the `Shuffle` procedure. The port at w leading to its parent v is set to 0, and the port at v leading to its child w is given by the `Shuffle` procedure. On the other hand, if w is inserted between v and its child u, then its light and heavy labels are set to be the light and ancestry labels at u, respectively. In addition, the corresponding port numbers at u and v remain the same, the port number at w leading to u is set to 1 (thus making u the heavy child of w), and the port to its parent is set to 0. In either case, the ancestry label of the inserted vertex w is given by Protocol `Dyn-Anc`.

Recall that the insertion of a leaf w triggers the simulation of Procedure `Shuffle`$(\{v, w\})$ at w's parent v. Apart from that, two other cases trigger the invocation Procedure `Shuffle`. Let us first describe the first of these two. Throughout the dynamic scenario, each vertex v in the tree T keeps a

counter $c_q(v)$, for every integer q (in fact, only $O(\log n)$ of these counters are non-zero at any given time). A *triggering event* occurs at a vertex u when one of its counters $c_q(u)$ becomes 2^{q+2}. In this case, we would like to invoke Procedure `Shuffle` on the subtree $T' = T_{light}(u)$. For that, we first invoke Procedure `Prepare_Subtree` at u, whose goal is to prepare the subtree T' for the invocation of Procedure `Shuffle`(T'), by freezing all nodes in the subtree T'. This can be implemented using $O(|T'|)$ messages, by a simple broadcast and upcast operation on T' (the value $|T'|$ used in the complexity of the procedure is calculated when the procedure is completed; we will guarantee that the procedure is indeed completed after finite time). When Procedure `Prepare_Subtree` is completed, Procedure `Shuffle`$(T_{light}(u))$ is initiated at u.

The third type of an event that triggers an invocation of Procedure `Shuffle` is when some leaf u joins the tree with a light label that satisfies $|L_{light}(u)| > 2^5 \log n'_0$, where n'_0 is the number of nodes in the tree, when the last `Shuffle` procedure was invoked on the whole tree. This triggering event which occurs at a vertex u, will result in an invocation of Procedure `Shuffle`(T'), on some subtree T' whose root $\rho = \rho(u)$ is one of u's ancestors, called a *pivot* vertex. The specific choice of the pivot $\rho(u)$ guarantees that the chosen subtree T' is on the one hand unbalanced, and on the other hand, relatively small (but not too small as that may result in too many invocations of Procedure `Shuffle`.) Specifically, the *pivot* vertex $\rho(u)$ is the closest ancestor of u satisfying the following conditions.

The Pivot Conditions**:** $|L_{light}(\rho(u))| \leq 2^4 \log n'_0$ and either one of the following conditions hold.

(1) The child w of $\rho(u)$ which leads to u is light, and there exists a q such that (i) $Port(\rho(u), w) \in I_q$ and (ii) $\omega(w) > \omega^0_{light}(\rho(u))/2^{q-1}$, where $\omega^0_{light}(\rho(u))$ is the last record known to $\rho(u)$ of its light weight,

(2) There exists a light child w of $\rho(u)$ such that $\omega(w) > 4/5 \cdot \omega(\rho(u))$,

(3) $\rho(u)$ is the root of the whole tree T, and $\omega(\rho(u)) > 2n'_0$.

The structure of the subtree T', on which the next Procedure `Shuffle` is applied, depends on which of the pivot conditions its root $\rho(u)$ satisfies. In the case where the pivot $\rho(u)$ satisfies the first pivot condition, the subtree T' is $T_w \cup \{\rho(u)\}$. Otherwise, if the pivot $\rho(u)$ does not satisfy the first pivot condition (but satisfies either the second or the third pivot condition), then the subtree T' is $T_{\rho(u)}$.

When the above type of a triggering event occurs at a leaf u (i.e., when u joins with a light label that satisfies $|L_{light}(u)| > 2^5 \log n'_0$), Procedure `Find_Pivot`$(u)$ is invoked at u, whose goal is to find the pivot $\rho(u)$ and to freeze the nodes in the corresponding subtree T'.

Procedure `Find_Pivot`(u) can be implemented using $O(|T'|)$ messages, as follows (again, $|T'|$ is measured when the procedure is completed). First, the leaf u creates an *agent* (see e.g., [15]). This agent then starts climbing from u up the tree, freezing every node it visits. The agent, coming from a child w to its parent v, informs v of w's weight. When the agent reaches some vertex v for the first time (coming from its child w), it checks whether the first pivot condition holds at v. If the condition is satisfied then Procedure `Find_Pivot`(u) is completed, and Procedure `Shuffle` is invoked on $T' = T_w \cup \{v\}$. Otherwise, a broadcast and upcast operation is performed on the subtree $T_v \setminus T_w$. The broadcast freezes the vertices in $T_v \setminus T_w$, and the upcast lets each vertex $z \in T_v \setminus T_w$ know it weight. Subsequently, v checks whether it satisfies either the second or third pivot conditions. If it does, then the procedure is completed and Procedure `Shuffle` is invoked on $T' = T_v$, and otherwise, the agent continues to v's parent. (It turns out, that if the agent reaches the root, then the root must be a pivot.)

In order to avoid collisions between the applications of different procedures, we do the following. First, if some procedure S of type either `Find_Pivot` or `Prepare_Subtree` tries to enter a vertex which initiated a `Shuffle` procedure that hasn't been completed yet, then Procedure S first waits for the `Shuffle` procedure to be completed and only then enters v and continues its action. Second, regarding collisions between different procedures of types either `Find_Pivot` or `Prepare_Subtree`, when two (or more) such procedures meet at a vertex v, only one of them continues from v. To save on messages, the 'winning' procedure will use the outcome of the other procedures instead of redoing their work. The procedure that wins in the 'competition' is the one coming to v from its parent, if indeed one comes from there, and otherwise, it is the leftmost one of them, i.e., the one coming from the leftmost child of v. The formal description of how to implement the above 'traffic rules' is rather straightforward, and is therefore deferred to the full paper.

Note that the protocol that grants permits to requests is Protocol `Dyn-Anc`. Therefore, it may happen that while some Procedure `Find_Pivot` or `Prepare_Subtree` tries to 'catch' a subtree and 'freeze' it, vertices are continuing to join the subtree and the procedure is never completed. Fortunately, Protocol `Dyn-Anc` operates in iterations, and in each iteration the number of nodes is finite. Moreover, between iterations, the tree T remains fixed. Therefore, to guarantee that each procedure is completed eventually, when an iteration of Protocol `Dyn-Anc` ends, Protocol `Dyn-Rout` performs a 'sweep' on the tree, which makes sure that before the next iteration starts, all procedures are completed.

We are now ready to describe Procedure `Shuffle`.

Procedure `Shuffle`. As mentioned, for balancing the tree, Protocol `Dyn-Rout` occasionally invokes Procedure `Shuffle` on different subtrees. (In particular, in the preprocessing stage, before the scenario actually starts, Procedure `Shuffle` is invoked on the whole initial tree for initializing the labels and port numbers.) Procedure `Shuffle`, when invoked on a subtree T' reorganizes it and makes it consistent with the whole tree. In the reorganization, the vertices in T' are assigned new labels and port numbers, and may also replace their heavy children, i.e., assign the port number 1 to a different incident port. The operation of Procedure `Shuffle` on T', called Procedure `Shuffle`(T'), varies according to the structure of T'. Let us first describe the operation of Procedure `Shuffle`(T') in the case where $T' = T_\rho$.

It follows from the description of the update protocol that as long as Procedure `Shuffle`(T') is operating, the subtree T' remains fixed. We assume that when Procedure `Shuffle`(T') starts, all the vertices in T' are already assigned labels and port numbers. (The initial invocation of Procedure `Shuffle`(T), that occurs in the preprocessing stage, starts with empty labels and arbitrary port numbers.) The assignment of new labels and port numbers to the nodes in T' is made carefully, in order to keep the scheme correct at all times. For that, we introduce four other sublabels at each label (that did not exist in the static scheme of [10]) called the *port* label, the *future* label, the *first bit* label, and

the *busy* label. The future label at each vertex w contains two sublabels, namely, the *future light* sublabel $f\mathrm{L}_{light}(w)$, and the *future heavy* sublabel $f\mathrm{L}_{heavy}(w)$.

Procedure `Shuffle`(T') is initiated at the root ρ of T'. At the first stage, a broadcast and upcast operation in performed on T'. This operation guarantees that upon its completion, the following holds.
(1) Each non-leaf vertex $w \in T'$ knows its port number leading to one of its children $f\mathrm{H}(w)$ with maximal weight. The ancestry label of $f\mathrm{H}(w)$ as well as the corresponding port number at w are stored in the future heavy sublabel of w. The vertex $f\mathrm{H}(w)$ is called a *future heavy* vertex, and all other children of w are called *future light*. The *future light subtree* of w, denoted $f\mathrm{T}_{light}(w)$, is the subtree $T_w \setminus T_{f\mathrm{H}(w)}$, and the *future light weight* of w, denoted $f\omega_{light}(w)$, refers to the number of nodes in $f\mathrm{T}_{light}(w)$, i.e., $f\omega_{light}(w) = |f\mathrm{T}_{light}(w)|$.
(2) Each vertex $w \in T'$, knows its weight $\omega(w)$ as well as its future light weight $f\omega_{light}(w)$.
(3) The counters at every vertex $w \in T'$ are all zero.

Subsequently, after the broadcast and upcast operation is completed, the root ρ of T' creates an agent, that performs a DFS tour in the subtree T'. The DFS tour is performed by first visiting the children with higher port numbers (in particular, the heavy child is visited last).

The busy label of a node consists of a single bit, informally indicating whether the DFS tour has reached the node or not; specifically, when the agent reaches a node for the first time, it sets its busy label to 1.

Whenever the agent is at a vertex $w \in T'$ and it is on its way to one of w's future light children z, it sets the port label $\mathrm{L}_{port}(w)$ at w as follows. $\mathrm{L}_{port}(w)$ contains two sublabels, namely, the *current port* sublabel $c\mathrm{L}_{port}(w)$, which is set to be $Port(w, z)$, and the *future port* sublabel $f\mathrm{L}_{port}(w)$, which is chosen as follows. For every two numbers $a < b$, let $[a, b)$ denote the set of integers $a \leq i < b$. For every two integers q and m, let $I_q = [2^{q+2}, 2^{q+3})$ and let $J_q(m) = [\frac{m}{2^{q+1}}, \frac{m}{2^q})$. Before continuing its DFS tour to the future light child z, the agent first sends a message to z, asking it for its weight $\omega(z)$. The agent (which is at w) now finds the value q such that $\omega(z) \in J_q(f\omega_{light}(w))$. It then selects $2^{q+2} + c_q(w)$ to be the future port sublabel at w, and then raises $c_q(w)$ by 1.

When the agent returns to w from one of its future light children z, is sets its corresponding port number (leading to z) to be the port in its future port sublabel. Note that for every q and every future light child z of w, such that $\omega(z) \in J_q(f\omega_{light}(w))$, we have $\omega(z) \geq f\omega_{light}(w)/2^{q+1}$. Thus, throughout the execution of this `Shuffle` procedure, $c_q(w) < 2^{q+1}$, and therefore the value $2^{q+2} + c_q(w)$ is in the range $[2^{q+2}, 2^{q+2} + 2^{q+1}) \subset I_q$. This guarantees that the port numbers assigned to w by the procedure are disjoint (since the intervals I_q are disjoint). However, we still need to make sure that these *new* port numbers are different also from the existing *old* port numbers at w, which were assigned before the procedure started. For that, we maintain the invariant, that at any given time throughout the scenario, the least significant bit in the binary strings of all the older port numbers at w leading to its light children is the same. This bit, called $\beta(w)$, is encoded in the *first bit* label of w. When the agent wishes to assign a new port sublabel to w, it first encodes $\bar{\beta}(w)$ in the least significant bit of that string, and then concatenates it with the string obtained according to the method above. This guarantees that all (new) port numbers assigned by the procedure at w are disjoint also from the (older) ones assigned before the procedure started. We would like to point out that the DFS tour of the agent in T' is made according to the order of the older port numbers only.

When the agent arrives at a vertex $w \in T' \setminus \{\rho\}$ for the first time (coming from its parent $p(w)$), it sets the future light sublabel at w as follows. If $w = f\mathrm{H}(p(w))$ then w's future light sublabel is set to be the same as the future light label of its parent. Otherwise, if $w \neq f\mathrm{H}(p(w))$, then its future light sublabel is set to be the future light label of its parent concatenated with the parent's new port sublabel, i.e., $f\mathrm{L}_{light}(w) = f\mathrm{L}_{light}(p(w)) \circ f\mathrm{L}_{port}(p(w))$. (These values, $f\mathrm{L}_{light}(p(w))$ and $f\mathrm{L}_{port}(p(w))$, are curried by the agent while moving from $p(w)$ to w.)

When the agent returns to $w \in T'$ after visiting all its children, it does the following. (1) If $w \neq \rho$, it sets the light label at w to be w's future light sublabel (otherwise, if $w = \rho$, the light label at w remains the same), (2) sets the heavy label at w to be the ancestry label of $f\mathrm{H}(w)$, which is stored in its future heavy sublabel, (3) sets the port number at w leading to $f\mathrm{H}(w)$ to be 1 (recall, the older port number at w leading to $f\mathrm{H}(w)$, is encoded in w's future heavy sublabel), (4) empties the port and future labels at w, (5) sets the first bit label at w to be $\bar{x}$, where x is the previous first bit label at w.

When the agent returns to ρ after visiting all the nodes in T', it get canceled. Subsequently, ρ initiates a broadcast operation for making all vertices in T' *unfrozen*, and for setting the busy label at each node in T' to 0. This completed the description of Procedure `Shuffle`(T'), in the case where $T' = T_\rho$.

The subtree T' can take two other forms. The first form is when $T' = T_{light}(\rho)$, i,e., T' contains ρ, all its light children and all their descendants, and the second form is when T' contains ρ and just one of its light children u, together with all its descendants, i.e., $T' = T_u \cup \{\rho\}$. The operation of Procedure `Shuffle`(T') in these two cases is rather similar to the case where $T' = T_\rho$, except for the following changes.

First, in the above two cases, the future heavy child of ρ, namely, $f\mathrm{H}(\rho)$, is simply chosen to be the heavy child $\mathrm{H}(\rho)$. (We do not choose a new heavy child for ρ, since, in these cases, during the time the procedure operates, the root ρ does not necessarily know its precise weight $\omega(\rho)$. Letting ρ know its weight may cause the procedure to send too many messages, as the subtree T_ρ might be much larger than T'.)

The remaining changes in the description of the procedure apply only for the case where $T' = T_u \cup \{\rho\}$, where u is one of ρ's light children. First, in this case, it is not guaranteed that ρ knows its future light weight $f\omega_{light}(\rho)$. Therefore, while applying the above method for choosing the port label at ρ, the procedure uses ρ's last known value of of its light weight, denoted $\omega^0_{light}(\rho)$, instead of using its future light weight $f\omega_{light}(\rho)$. Second, in this case, Procedure `Shuffle`(T') does not initializes the counters at ρ to zero. Therefore, when the procedure raises the corresponding existing counter $c_q(\rho)$ by 1, it may cause the corresponding port number at ρ to be outside of I_q, which may violate the requirement that the port numbers at ρ must be disjoint. Therefore, whenever a counter $c_q(\rho)$ reaches 2^{q+2}, the agent (which is at ρ) is canceled and the `Shuffle` procedure is completed. (As described before, such a phenomena

triggers the invocation of Procedure `Shuffle` on a subtree containing T_ρ.) Finally, note that in this case, the procedure replaces only one port number at ρ (the one leading to u). Therefore, to maintain the 'first bit invariant', we use $\beta(\rho)$ as the least significant bit in the port label at ρ, instead of its inverse bit $\bar{\beta}(\rho)$.

The message complexity of Procedure `Shuffle`(T') is $O(|T'|)$, where $|T'|$ is calculated when the procedure is initiated (recall, the subtree T' remains fixed throughout the operation of the procedure). This completes the description of Procedure `Shuffle`. We are now ready to describe the router of the scheme.

The router. Given the labels of any two vertices u and v, the router needs to find the current port number at u leading to the next vertex w on the shortest path connecting u and v. Using the ancestry labels and the decoder of Protocol `Dyn-Anc`, the router can find whether u is an ancestor of v or not. In the case where v is not a descendant of u, the router outputs 0 (which is the port number at u leading to u's parent). If v is a descendant of $\text{H}(u)$ then this is detected using the heavy label at u and the ancestry label at v. In this case, the router outputs 1.

If the busy label at u is empty (which would mean, informally, that no agent of some `Shuffle` procedure that is currently operating on a subtree containing u, has entered u), then the router outputs the port obtained by comparing the light labels of u and v, using Extract.

Consider now the case that the busy label at u is not empty (this means, informally, that such an agent reached u).

If v is a descendant of $f\text{H}(u)$ then this is detected using the future heavy sublabel at u and the ancestry label at v. In this case, the router outputs the port number stored in the future heavy sublabel of u.

Assume now that v is a descendant of u but not of neither $\text{H}(u)$ nor $f\text{H}(u)$. In this case, if the future light sublabel at v is not empty (which would mean that the agent is currently in T_v), then the router outputs the port number stored in the old port sublabel at u. Otherwise, if the future light sublabel at v is empty, then consider the following. If the busy label at v is 0 (which would mean that the agent hasn't reached v yet), then the router outputs the port number obtained by using Extract on the corresponding light labels. If, on the the other hand, the busy label at v is 1, then let $Port$ be the port number obtained by comparing the light label at v with the future light sublabel at u, using Extract. Consider two cases. If $Port$ is the port number in the new port sublabel at u (which would mean that the agent has reached v but hasn't returned to u after visiting v), then the router outputs the old port sublabel at u. Otherwise, (this corresponds to the case where the agent returned to u from w after visiting v, but hasn't visited all of u's children yet), the router simply outputs the port number $Port$.

4.1.1 Analysis

The proofs of the following three claims are deferred to the full paper.

Claim 4.1. *For any time and any vertex u, the value of each counter $c_q(u)$ is at most 2^{q+2}.*

Claim 4.2. *Each procedure is eventually completed.*

Claim 4.3. *Every frozen vertex eventually becomes unfrozen again.*

Lemma 4.1. *Protocol `Dyn-Rout` implements a routing scheme which is correct at all times.*

Proof. The fact that each requested topology change eventually occurs results from the fact that Protocol `Dyn-Anc` eventually grants a permit to the request, and from Claim 4.3. It is now left to show that the decoder is correct at any given time. I.e., given the labels of any two vertices u and v, we need to show that the router outputs the current port number at u leading to the next vertex w on the shortest path connecting u and v.

If v is not a descendant of u then the next vertex on the shortest path connecting u and v is u's parent. Indeed, since this case is detected by the router at u (using the ancestry labels of u and v and the decoder of Protocol `Dyn-Anc`), it outputs 0, which is the port leading from u to its parent. If v is a descendant of $\text{H}(u)$ then similarly, this is detected by using the heavy label at u and the ancestry label at v. In this case, the router outputs 1, which is the port number at u leading to $\text{H}(u)$.

If the busy label at u is empty, then u was not visited by an agent of a `Shuffle` procedure, that is currently operating on a subtree containing u. Therefore, the light label at u is a prefix of the light label at v, and the desired port number $Port(u, w)$ is found by comparing the light labels of u and v, using Extract.

Consider now the case that the busy label at u is not empty, and let $\mathcal{S}$ be the `Shuffle` procedure that is currently operating on a subtree containing u (clearly, the subtree also contains v). Note that the agent of $\mathcal{S}$ has already entered u. If v is a descendant of $f\text{H}(u)$ then similarly to before, this is detected using the ancestry label of $f\text{H}(u)$, which is stored in the heavy label at u, and the ancestry label at v. In this case, the router outputs the port number stored at the future heavy sublabel at u, which is the port at u leading to $f\text{H}(u)$.

The more interesting case is when v is a descendant of one of u's children but not of neither $\text{H}(u)$ nor $f\text{H}(u)$. If the future light label at v is not empty, then this means that the agent of $\mathcal{S}$ is currently in T_v. Therefore, in particular, the agent has left u via port number $Port(u, w)$ and hasn't returned from there to u yet. In this case, the router outputs the port number stored in the old port sublabel at u, which is indeed the port number of the port at u leading to w, the last child of u visited by the agent.

If, on the other hand, the future light label at v is empty, then consider the following.

If the busy label at v is 0 then the agent of $\mathcal{S}$ hasn't reached v yet. In particular, the agent hasn't returned to u, and therefore the light labels at both u and v haven't been modified by $\mathcal{S}$ yet. In this case, the router outputs the port number obtained using Extract on the corresponding light labels, which is therefore indeed, $Port(u, w)$.

If, on the the other hand, the busy label at v is 1, then let $Port$ be the port number obtained by comparing the light label at v with the future light sublabel at u, using Extract. Consider two cases. If $Port$ is the port number in the new port sublabel at u then this means that the agent is on its way from v to u, and therefore, $Port(u, w)$ is encoded in the old port sublabel at u. Note, the port number in the old port sublabel at u is exactly what the router outputs in this case. Otherwise, if $Port$ is not the port number in the new port sublabel at u then this means that the agent returned to u from w (but hasn't visited all of u's children

yet). In this case, $Port(u,w)$ is obtained by comparing the light label at v with the future light sublabel at u, using EXTRACT. I.e., $Port(u,w) = Port$, which is exactly what the router outputs, in this case. □

LEMMA 4.2. *The label size of Protocol* `Dyn-Rout` *is* $O(\log n)$.

PROOF. Since protocol `Dyn-Anc` guarantees labels of size $O(\log n)$, we get that the ancestry labels are encoded using $O(\log n)$ bits. Clearly, each first bit label and each busy label is encoded using $O(1)$ bits.

Let us now show that each port number is encoded using $O(\log n)$ bits. Fix a vertex v and a port p leading from v to one of its children w. Consider the time t when the current port number of p was assigned. At that time, the port number of p was equal to $2^{q+2} + c_q(v)$, for some q such that $\omega(w) < \frac{n}{2^q}$. This implies that $2^q < n$. From Claim 4.1, the counter $c_q(v)$ is always at most 2^{q+2}, and therefore, we get that the port number of p is encoded using $O(\log n)$ bits. It follows that the port label as well as the future heavy sublabel are encoded using $O(\log n)$ bits.

It is left to show that the light labels and future light labels are also encoded using $O(\log n)$ bits. Since each light label was once a future light label, it is sufficient to bound the number of bits in a future light label.

The future light label of a vertex v is assigned only when Procedure `Shuffle` is invoked on a subtree T' containing v. Consider such a `Shuffle` procedure $\mathcal{S}$ and let ρ be the root of T'. The procedure $\mathcal{S}$ may be invoked only when one of the following three events occur. (1) The root ρ of T' was found to be a pivot vertex by Procedure `Find_Pivot`, (2) a `Shuffle` procedure $\mathcal{S}'$ was invoked on the subtree $T_w \cup \{\rho\}$, where w is one of ρ's light children, and consequently, one of the counters $c_q(\rho)$ exceeded its threshold, (3) a leaf u joined the tree as a child of ρ, and $\mathcal{S}$ is the `Shuffle` procedure that was invoked on ρ and u.

Consider first the case where one of the first two events above occur. In this case, it can be shown that the light label at ρ is encoded using at most $2^4 \log n$ bits. A future light label in T' that is assigned by $\mathcal{S}$ is composed of the light label at ρ concatenated with a sequence of new port numbers. Using similar arguments as the ones mentioned in the proof of the static routing scheme in [10], we get that the number of bits used for encoding the concatenated port numbers is at most $2^4 \log n$. Altogether, the number of bits in a future light label which is given by $\mathcal{S}$ is thus at most $2^5 \log n$.

Consider now the case, that some leaf u is added as a child of ρ and $\mathcal{S}$ is the `Shuffle` procedure that was invoked on ρ and u (this procedure was simulated at ρ). Note, that the (future) light label at u is encoded using at most $|L_{light}(\rho)| + \log n$ bits. If the `Shuffle` procedure $\mathcal{S}'$ that assigned ρ its current light label was invoked due to either one of the first two events above, then we know that $|L_{light}(\rho)| \leq 2^5 \log n$, and therefore, $|L_{light}(u)| \leq (2^5+1)\log n$. Otherwise, ρ was added as a child of a leaf ρ' and $\mathcal{S}'$ was the `Shuffle` procedure that was invoked on ρ' and ρ. Assume, by contradiction, that S' assigned ρ a light label which consists of more than $2^5 \log n$ bits. In this case, before u was added, ρ would have initiated a `Find_Pivot` procedure. That procedure would have resulted in a `Shuffle` procedure on a subtree containing ρ, which, before u is added, would assign ρ a light label that consists of at most $2^5 \log n$ bits. This contradicts our assumption. We therefore get that the light label of ρ consists of at most $2^5 \log n$ bits, and consequently, $|L_{light}(u)| \leq (2^5+1)\log n$. The lemma follows. □

LEMMA 4.3. *The message complexity of Protocol* `Dyn-Rout` *is* $O(n \log n)$.

PROOF. By Theorem 3.1, the number of messages resulted from applying Protocol `Dyn-Anc` is $O(n \log n)$. It remains to bound the number of messages resulted from the different applications of procedures `Find_Pivot`, `Prepare_Subtree` and `Shuffle`. Clearly, this number is asymptotically bounded by the number of messages resulted from the different applications of procedure `Shuffle`.

First note, that the simulated `Shuffle` procedures (that are invoked before inserting a leaf) do not incur any message complexity. Second, observe that the `Shuffle` procedures that are invoked at a pivot vertex which satisfies the third pivot condition incur $O(n)$ message complexity in total.

Let Γ be the collection of all other `Shuffle` procedures that where invoked throughout the dynamic scenario. We now bound the message complexity resulted from the procedures in Γ. For every `Shuffle` procedure $\mathcal{S} \in \Gamma$, let $T(\mathcal{S})$ denote the subtree on which $\mathcal{S}$ is applied. Recall that the message complexity of $\mathcal{S}$ is $O(|T(\mathcal{S})|)$, calculated at the time the procedure takes place. The total number of messages incurred by the procedures in Γ is therefore $O(\sum_{\mathcal{S}\in\Gamma} |T(\mathcal{S})|)$.

At any given time, let $\mathcal{LC}$ denote the collection of light vertices in T. For every vertex v, let

$$\omega_{light-child}(v) \leftarrow \begin{cases} \omega(v)\ , & \text{if } v \in \mathcal{LC} \\ 0\ , & \text{otherwise.} \end{cases}$$

At any given time, let $\omega^0_{light-child}(v)$ denote the value $\omega_{light-child}(v)$ when the last Procedure `Shuffle` was completed on a subtree containing T_v, and let $\omega^0_{light}(v)$ denote the value $\omega_{light}(v)$ when the last time Procedure `Shuffle` was completed on a subtree containing $T_{light}(v)$.

Let us now examine the behavior of the value $\Psi = \sum_{v\in T}(\omega_{light}(v) - \omega^0_{light}(v)) + \sum_{v\in T}(\omega_{light-child}(v) - \omega^0_{light-child}(v))$ during the execution.

First suppose a vertex v is added to the tree. Since there are $l(v) = O(\log n)$ light edges on the path from v to the root, Ψ increases by $O(\log n)$.

Now suppose that a `Shuffle` procedure $\mathcal{S} \in \Gamma$ is completed at some time t. Let ρ be the root of $T(\mathcal{S})$. Our goal now is to show that the invocation of $\mathcal{S}$ caused Ψ to decrease by $\Omega(|T(\mathcal{S})|)$. First note, that for every vertex v, the invocation of $\mathcal{S}$ may only increase the values $\omega^0_{light}(v)$ and $\omega^0_{light-child}(v)$, and therefore, the values $(\omega_{light}(v) - \omega^0_{light}(v))$ and $(\omega_{light-child}(v) - \omega^0_{light-child}(v))$ can only be deceased.

Consider first the case that $\mathcal{S}$ was invoked because one of the counters at ρ exceeded its threshold. Let t' be the time $\mathcal{S}$ was invoked. At time t', we have $c_q(\rho) = 2^{q+2}$ for some q. Let t_0 be the last time before time t, that some application of Procedure `Shuffle` on a subtree containing $T_{light}(\rho)$ was completed. At time t_0, we have $c_q(\rho) \leq 2^{q+1}$. Note, from time t_0 until time t, no Procedure `Shuffle` was operating on a subtree containing T_ρ. Therefore, during that time period, ρ did not change its heavy child. Notice also, that during that time period, if the port at ρ leading to one of its children u, received a new value $Port(\rho,u)$ in I_q then, from that point in time until time t', if the number of the port changed again, then it changed only to some value in $I_{q'}$ where $q' < q$.

This means that if at some point during the time period between time t_0 and time t', the value of $c_q(\rho)$ was c then at that time, there were at least $c \cdot \omega^0_{light}(\rho)/2^{q+1}$ vertices in $T_{light}(\rho) = T(\mathcal{S})$. Since at time t', we had $c_q(\rho) = 2^{q+2}$, then at that time, $\omega_{light}(\rho) \geq 2^{q+2} \cdot \omega^0_{light}(\rho)/2^{q+1} = 2 \cdot \omega^0_{light}(\rho)$. Since the weight of ρ cannot decrease with time, we get that also just before time t, $\omega_{light}(\rho) \geq 2 \cdot \omega^0_{light}(\rho)$. However, at time t, the value $\omega^0_{light}(\rho)$ increased and became $\omega_{light}(\rho)$. This means that at time t, the value $\omega_{light}(\rho) - \omega^0_{light}(\rho)$ decreased by $\Omega(\omega_{light}(\rho)) = \Omega(|T(\mathcal{S})|)$.

Now suppose that $\mathcal{S}$ was invoked at a pivot ρ that satisfied the first pivot condition. In this case, $T(\mathcal{S})$ is the subtree $T_w \cup \{\rho\}$, where w is the (light) child of ρ on the way to u. We now show that the value $\omega_{light-child}(w) - \omega^0_{light-child}(w)$ decreased at time t by $\Omega(|T(\mathcal{S})|)$. Let t_0 be the last time before time t that a `Shuffle` procedure was completed on a subtree containing T_w, and let t' the time when $\mathcal{S}$ was invoked. At time t', we have $\omega(w) > \omega^0_{light}(\rho(u))/2^{q-1}$ for q such that $Port(\rho(u), w) \in I_q$. On the other hand, for the same q, we have that at time t_0, $\omega(w) < \omega^0_{light}(\rho(u))/2^q$. This means that from time t_0 until just before time t, the weight of w increased by $\Omega(\omega^0_{light}(\rho(u))/2^{q-1}) = \Omega(\omega^0_{light-child}(w))$. Therefore, the value $\omega_{light-child}(w) - \omega^0_{light-child}(w)$ decreased at time t by $\Omega(\omega_{light-child}(w)) = \Omega(\omega(w)) = \Omega(|T(\mathcal{S})|)$.

Now suppose that $\mathcal{S}$ was invoked at a pivot ρ that satisfied the second pivot condition. Using similar arguments, it follows that at time t, the value $\omega_{light}(\rho) - \omega^0_{light}(\rho)$ decreased by $\Omega(\omega_{light}(\rho)) = \Omega(|T(\mathcal{S})|)$.

Combining the above observations, we get that

$$0 \leq \Psi \leq O(n \log n) - \Omega \sum_{\{\mathcal{S} | \mathcal{S} \in \Gamma\}} (|T(\mathcal{S})|).$$

This yields that $\sum_{\{\mathcal{S} | \mathcal{S} \in \Gamma\}} (|T(\mathcal{S})|) = O(n \log n)$, which establishes the lemma.

□

The following theorem now follows by combining Lemma 4.1, Lemma 4.2, and Lemma 4.3.

THEOREM 4.4. *Protocol* `Dyn-Rout` *implements a routing scheme on a growing tree, which is correct at all times. The label size of the scheme is* $\Theta(\log n)$ *and its message complexity is* $O(n \log n)$.

4.2 Supporting deletions of nodes

We now extend Protocol `Dyn-Rout` to support also deletions of nodes of degree at most 2. The idea is similar to the one used in Theorem 3.3. The detailed proof of the following theorem is deferred to the full paper.

THEOREM 4.5. *There exists a dynamic compact routing scheme on trees, supporting insertions of leaves and internal vertices as well as removals of vertices of degree at most 2, using message complexity* $O(n_0 \log^2 n_0) + O(\sum_i \log^2 n_i)$, *where* n_i *is the number of nodes in the tree, after the* i*'th topological change.*

5. REFERENCES

[1] Y. Afek, B. Awerbuch, S.A. Plotkin and M. Saks. Local management of a global resource in a communication network. *J. ACM*, 43:1–19, 1996.

[2] I. Abraham, and C. Gavoille. Object location using path separators. In *PODC 2006*.

[3] Y. Afek, E. Gafni, and M. Ricklin. Upper and lower bounds for routing schemes in dynamic networks. In *FOCS 1989*, 370–375.

[4] S. Abiteboul, S. Alstrup, H. Kaplan, T. Milo and T. Rauhe. Compact labeling schemes for ancestor queries. *SIAM Journal on Computing* **35**, (2006), 1295–1309.

[5] Y. Afek and M. E. Saks. Detecting Global Termination Conditions in the Face of Uncertainty. In *PODC 1987*.

[6] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton and J. Zito. Two Simplified Algorithms for Maintaining Order in a List. In *ESA 2002*.

[7] R. Bar-Yehuda and S. Kutten. Fault tolerant distributed majority commitment. *J. Algorithms*, 9(4):568–582, 1988.

[8] P. Dietz and D. Sleater. Two Algorithms for Maintaining Order in a List. In *STOC 1987*, 365–372.

[9] Y. Emek and A. Korman. Estimating the Number of Events in a Network: an Online Distributed problem, 2008.

[10] P. Fraigniaud and C. Gavoille. Routing in trees. In *ICALP 2001*, 757–772.

[11] C. Gavoille, D. Peleg, S. Pérennes and R. Raz. Distance labeling in graphs. *J. of Algorithms*, **53(1)**, (2004) 85–112.

[12] S. Kannan, M. Naor, and S. Rudich. Implicit representation of graphs. In *SIAM J. on Descrete Math* **5**, (1992), 596–603.

[13] H. Kaplan, T. Milo and R. Shabo. A Comparison of Labeling Schemes for Ancestor Queries. In *SODA 2002*.

[14] A. Korman. General compact labeling schemes for dynamic trees. *J. Distributed Computing*, 20(3):179–193, 2007.

[15] A. Korman and S. Kutten. Controller and estimator for dynamic networks. In *PODC 2007*.

[16] A. Korman and D. Peleg. Compact Separator Decomposition for Dynamic Trees and Applications. *J. Distributed Computing*, to appear.

[17] A. Korman, D. Peleg, and Y. Rodeh. Labeling schemes for dynamic tree networks. *Theory Comput. Syst.*, 37(1):49–75, 2004.

[18] A. Korman and D. Peleg. Labeling schemes for weighted dynamic trees. *J. Information and Computation*, 205(12):1721–1740, 2007.

[19] A. Korman and D. Peleg. Dynamic routing schemes for graphs with low local density. *ACM Trans. on Algorithms*, to appear.

[20] N. Santoro and R. Khatib. Labelling and implicit routing in networks. *The Computer Journal* **28**, (1985), 5–8.

[21] M. Thorup and U. Zwick. Compact routing schemes. In *SPAA 2001*, 1–10.

[22] J. Van Leeuwen and R. B. Tan. Interval routing. *The Computer Journal* **30**, (1987), 298–307.