

# Compact Routing Schemes for Dynamic Trees in the Fixed Port Model

Amos Korman \*

CNRS and Université Paris Diderot - Paris 7, France.  
amos.korman@gmail.com

**Abstract.** The current paper considers the routing problem in dynamic trees under the *fixed-port* model, in which an adversary chooses the port numbers assigned to each node. We present two routing schemes for dynamic trees that maintain labels of asymptotically optimal size using extremely low average message complexity (per node). Specifically, we first present a dynamic routing scheme that supports additions of both leaves and internal nodes, maintains asymptotically optimal labels and incurs only  $O(\log^2 n / \log^2 \log n)$  average message complexity. This routing scheme is then extended to support also deletions of nodes of degree at most 2. The extended scheme incurs  $O(\log^2 n)$  average message complexity and still maintains asymptotically optimal labels.

We would like to point out that the best known routing scheme for dynamic trees that maintains asymptotically optimal labels in the fixed port model has very high average message complexity, namely,  $O(n^\epsilon)$ . Moreover, that scheme supports additions and removals of leaf nodes only.

**Keywords:** Distributed algorithms, dynamic networks, routing schemes, trees.

## 1 Introduction

**Background.** The study of methods for designing *routing schemes* in *static* (fixed topology) settings is rather well developed. Typically, the main goal of such a scheme is to equip each node with a (hopefully short) label, such that efficient routing can be performed between any two nodes, based merely on the label of the destination node and the labels of intermediate nodes on the path. In particular, a classical *routing problem* for a family of graphs  $\mathcal{F}$  consists of a method for assigning a *label* to each node in each graph  $G \in \mathcal{F}$ , such that given the label of any node  $v$  in some given graph  $G \in \mathcal{F}$  and the label of any destination node  $u \in G$ , node  $v$  can find which of its incident port numbers (in  $G$ ) leads to the next node on a shortest path connecting  $v$  and  $u$ . Most works on such routing schemes evaluate the scheme by its *label size*, i.e., the maximal number of bits stored in a label (see e.g., [2, 3, 5, 15, 16]). Not surprisingly, the main objective was to construct *compact* schemes, which enjoy asymptotically optimal label size.

The quality of a routing scheme strongly depends on the model considered. In particular, it is known that the label size of a routing scheme for trees depends on who

---

\* Supported in part by the ANR project ALADDIN, by the INRIA project GANG, and by COST Action 295 DYNAMO.

has the power to assign the port numbers of the nodes. Specifically, two main models regarding port numbers are considered in the literature. For the *designer* port model, in which the designer of the routing scheme can freely select the port numbers assigned to each node (as long as they remain distinct), [5, 16] independently constructed a routing scheme with label size  $\Theta(\log n)$ , for the family of  $n$ -node (static) trees. Under the *fixed port* model, in which the port numbers (encoded using  $O(\log n)$  bits) are fixed by an adversary, [5] constructed a routing scheme on (static)  $n$ -node trees using labels of  $O(\frac{\log^2 n}{\log \log n})$  bits. This bound was shown to be asymptotically optimal in [6]. Though it requires larger labels, the fixed port model (considered also in this paper) may be found more useful in certain applications, as it allows for more modular constructions; in this context, the adversary, choosing the port numbers, models constraints on port numbers which are given by other protocols running on the network and using the same ports.

In contrast to 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 the topology changes, which may require the delivery of information from place to place. This raises the natural problem of maintaining short labels (hopefully asymptotically optimal) using low communication cost. Specifically, in addition to its label size, we evaluate a dynamic routing scheme by its *average message complexity*, i.e., the ratio of the number of messages sent during the execution to the number of nodes. This measure reflects the amount of work a typical node exerts for communication purposes.

The problem of maintaining a routing scheme in a dynamic tree was originally introduced by [4] (and implicitly in [1]). This problem was later studied in a series of papers [7, 8, 10, 11], which proposed different tradeoffs between the label size and the message complexity. Still, the best known compact routing scheme on dynamic trees in the fixed port model<sup>1</sup> has very high average message complexity, namely,  $O(n^\epsilon)$  [7]. Moreover, the types of topology changes supported by previous schemes on (fixed port) trees are limited to additions and removals of leaves only. On top of that, the correctness of previous such schemes is guaranteed only for *quiet* times, in which all updates concerning the previous topology changes, have already occurred.

**Our contribution.** In this paper we consider the fixed port model and present two compact routing schemes for dynamic trees that incur extremely low average message complexity.

The main contribution of the paper is the construction of an efficient compact routing scheme for growing trees, i.e., trees that undergo topology changes which are restricted to additions of both leaves and internal nodes. This routing scheme (given in Section 3) assigns and maintains asymptotically optimal  $\Theta(\frac{\log^2 n}{\log \log n})$ -bit labels, using  $O(\frac{\log^2 n}{\log^2 \log n})$  average message complexity, where  $n$  denotes the current tree size.

---

<sup>1</sup> In the dynamic setting, the fixed port model assumes that, at any given time, the port numbers (assigned by the adversary) are encoded using  $O(\log n)$  bits, where  $n$  is the maximal tree size until the given time. Moreover, once a port number is assigned, it remains fixed, and cannot be changed again (unless the port itself is removed).

Our scheme for growing trees can be extended to support also removals of degree 1 and 2 nodes using the method mentioned in [9]. For any given time, let  $n$  denote the maximal tree size until the given time. The extended scheme maintains  $\Theta(\frac{\log^2 n}{\log \log n})$ -bit labels using  $O(\log^2 n)$  average message complexity (the average message complexity is taken with respect to all nodes existing in the tree, including deleted ones). Since the extended scheme is a straightforward application of the method in [9], its formal description is deferred to the appendix.

In contrast to the previous routing schemes on (fixed-port) dynamic trees, whose correctness is guaranteed only for 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 note that our dynamic schemes as well as the dynamic schemes in [7–11], 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 routing scheme can be expected to be both compact and correct at all times, if nodes are being inserted to the tree in a very rapid succession. In particular, if an adversary can continuously insert internal nodes very fast, then it can prevent any message from reaching its destination. In the case where only leaves may join the tree, our schemes can be extended to support spontaneous insertions of leaves, however, in this case, the correctness is only guaranteed for quiet times (similarly to [7, 10]).

Routing schemes	Paper [7]	Paper [10]	This paper (result 1)	This paper (result 2)
Label size	$\Theta(\frac{\log^2 n}{\log \log n})$	$O(\log^2 n)$	$\Theta(\frac{\log^2 n}{\log \log n})$	$\Theta(\frac{\log^2 n}{\log \log n})$
Average message complexity	$O(n^\epsilon)$	$O(\log^4 n)$	$O(\frac{\log^2 n}{\log^2 \log n})$	$O(\log^2 n)$
Types of topological changes	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 all times	at all times

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

**Related work.** Routing schemes on static trees were investigated in various papers e.g., [14, 17]. Compact routing labeling schemes on static trees were given in [5, 6, 16]. Specifically, for the *fixed* port model, [5] gave a routing scheme using labels of  $O(\frac{\log^2 n}{\log \log n})$  bits. This bound was shown to be asymptotically optimal in [6].

The problem of maintaining a routing scheme in a dynamic tree under the fixed-port model was originally introduced by [1, 4], and later studied in a series of papers [7, 10, 11], which proposed different tradeoffs between the label size and the message complexity. In [1], the authors implicitly describe a dynamic routing scheme on a dynamically growing tree (in which only leaves are allowed to join) using logarithmic average message complexity but huge labels, namely, the label size is  $O(n \log n)$ . In

[4] the authors gave an improved dynamic routing scheme on dynamically growing trees using labels of size  $O(\Delta \log^3 n)$  (where  $\Delta$  is the maximum degree in the tree) and logarithmic average message complexity. Again, if only leaves can join the tree, [7, 11] gave a dynamic routing scheme with logarithmic average message complexity and label size that is larger than the optimum by a factor of  $O(\log n)$ .

In the case where leaves can also be removed from the network, [10] gave a dynamic routing scheme with average message complexity  $O(\log^4 n)$  and label size that is larger than the optimum by a factor  $O(\log \log n)$ . So far, the best known *compact* routing scheme on (fixed port) trees [7] uses  $O(n^\epsilon)$  average message complexity, where  $0 < \epsilon < 1$  is some constant. Moreover, previous routing schemes on (fixed port) dynamic trees support additions and removals of leaves only, and are guaranteed to be correct only at quiet times.

Recently, an efficient compact routing scheme for dynamic trees under the designer port model was constructed in [8]. Our main dynamic scheme bears some similarities with that scheme. For example, both routing schemes use an efficient dynamic ancestry scheme to detect ancestry relations between nodes. Moreover, both dynamic routing schemes are based on adapting the principles of the corresponding static schemes in [5] to the more complex dynamic setting. However, the corresponding static schemes in [5] (the one for the designer port model and the one for the fixed port model) are quite different, and thus require different handling when adapted to the dynamic setting. Specifically, efficient maintenance of a routing scheme with short labels requires the maintenance of an implicit (balanced) decomposition of the tree that strongly depends on the label structure. Thus, the differences in the label structure between the two schemes impose differences in the protocol that maintains the labels, as it must also maintain a different implicit tree decomposition.

## 2 Preliminaries

The network topology is described by an undirected communication tree  $T = \langle V, E \rangle$ , where  $V$  is a set of nodes, representing processors, and  $E$  is a set of edges, representing bidirectional communication links. We consider the standard asynchronous point-to-point message-passing model. Specifically, the communication between nodes is made by exchanging messages over the edges. Each message sent over an edge arrives without errors, at an arbitrary but finite delay. We assume that the computations performed at a node are made instantaneously.

We consider the *fixed port* model, in which, for each node  $v$ , an adversary assigns disjoint port numbers pointing at  $v$ 's incident edges (the assignment is local, and is not necessarily consistent between two neighboring nodes). In the static setting, it is required that each port number is encoded using  $O(\log n)$  bits, where  $n$  is the number of nodes in the tree.

The ancestry relation in the tree  $T$  is defined as the transitive closure of the parent-hood relation (in particular, a node is an ancestor of itself). A node  $v$  is a *descendant* of node  $u$  if  $u$  is an ancestor of  $v$ . For a node  $v$ , let  $T(v)$  denote the subtree of  $T$  hanging down from  $v$  (including  $v$ ), and let  $\omega(v)$ , denote the number of nodes in  $T(v)$ , i.e.,  $\omega(v) = |T(v)|$ . We refer to  $\omega(v)$  as the *weight* of  $v$ .

**Static routing schemes on trees.** A static *routing scheme* on trees is composed of the following components.

- 1) A *marker* algorithm that given a tree  $T$  assigns a label  $L(v)$  to each node  $v \in T$ .
- 2) A *router* algorithm that given the labels  $L(w)$  and  $L(v)$  of two nodes  $w$  and  $v$  in  $T$ , outputs  $\text{Rout}(w, v)$ , where  $\text{Rout}(w, v)$  is the port number at  $w$  leading to the next node on the shortest path connecting  $w$  and  $v$ .

The most common measure to evaluate a static routing scheme is the *label size*, i.e., the maximum number of bits stored in a label. A routing scheme with asymptotically optimal label size is called *compact*. In our schemes, the labels given to the nodes may contain several fields (we use the symbol  $\circ$  to concatenate such fields). Clearly, one can distinguish between the different fields with the aid of a sequential data structure that does not increase the asymptotic size of a label (see e.g., [7]).

**The dynamic models.** We assume that we may initialize the labels and data structure of the initial tree, in a *preprocessing stage*, which is completed 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 node  $u$  is added as a child of an existing node  $v$ . **(2) Add internal node (between neighbors  $v$  and  $w$ ):** Edge  $e = (v, w)$  splits into two edges  $(v, u)$  and  $(u, w)$  for a new node  $u$ . If  $v$  was  $w$ 's parent, then  $u$  is a child of  $v$  and  $w$  is considered a child of  $u$ . **(3) Remove node of degree at most 2:** A (non-root) node  $v$  of degree at most two is deleted. If  $v$  was internal node, then its (only) child becomes the child of  $u$ 's parent.

In the *growing tree* model it is assumed that only the first two types of topology changes may occur, namely, Add-leaf and Add internal node. A *growing tree* consists of an initial tree that may change according to the growing tree model.

When a new leaf  $u$  is inserted as a child of an existing node  $v$ , the corresponding ports at  $v$  and  $u$  are assigned (by an adversary) a port-number, under the constraint that at any given moment, the port numbers at each node are distinct. Once an adversary assigns a port number, this number cannot be changed (unless the link is removed). We assume that at any given time  $t$ , the port numbers are encoded using  $O(\log n)$  bits, where  $n$  denotes the maximal tree size until time  $t$ . (Note that for a growing tree, this means that the port numbers are encoded using  $O(\log n)$  bits, where  $n$  is the current tree size.) We would like to point out that this requirement on the size of the port numbers is also used in the static scheme of [5], and is necessary for obtaining short labels, as port numbers must somehow be encoded in labels. Moreover, this requirement seems realistic, as the adversary typically encodes port numbers using  $O(\log \Delta)$  bits, where  $\Delta$  is the maximal degree of the node so far (in some cases, one may run the dynamic routing scheme on a dynamic spanning tree of a dynamic network; in this case the adversary would typically encode the port numbers using  $O(\log \Delta)$  bits, where the degree  $\Delta$  relates to the dynamic graph rather than to the dynamic spanning tree. However, the requirement holds in this case too, as  $\Delta < n$ ).

If a leaf is removed, the corresponding link is removed, and thus the corresponding port number (at the existing parent) is also removed. If an internal node  $u$  is removed, then its parent becomes the parent of its child, and the corresponding port numbers (at the parent and child of  $u$ ) remain the same.

For a routing scheme in a dynamic tree, the definition of a router remains the same, however, the marker algorithm changes to distributed *update protocol*, whose goal is to assign and maintain the labels to fit the requirements of the router.

We consider the *controlled* dynamic model (considered also in [1, 8–10, 13]), in which the topological changes do not occur spontaneously. Instead, when an entity wishes to cause a topological change at some node  $u$ , 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 and the implementations of the topology changes, see [8, 9].

For a dynamic routing scheme, we are interested in the following complexity measures.

The *label size* is the maximum number of bits in a label taken over any node  $v$ .

The *average message complexity* is the ratio of the number of messages sent during the execution to the number of nodes ever existing in the tree (including deleted nodes).

### 3 The compact routing scheme for a growing tree

In this section we consider the growing tree model and establish our compact routing scheme `Dyn-Rout` for a growing tree, which incurs  $O(\frac{\log^2 n}{\log^2 \log n})$  average message complexity. Let us first recall the compact ancestry scheme on growing trees given in Section 3 in [8].

**Theorem 1.** [8] *In the designer port model, there exists a dynamic compact ancestry scheme on a growing tree that incurs  $O(\log n)$  average message complexity. I.e., using  $O(\log n)$  average message complexity, the scheme maintains at each node  $w$  in the growing tree an ancestry label  $L_{anc}(w)$  of size  $O(\log n)$ , such that for any two nodes  $u$  and  $v$ , given the ancestry labels  $L_{anc}(u)$  and  $L_{anc}(v)$  only, one can detect whether  $u$  is an ancestor of  $v$  or not.*

Let us denote the ancestry scheme given in Theorem 1 by Protocol `Dyn-Anc`. As mentioned, Protocol `Dyn-Anc` operates in the designer port model where the port numbers at each node can be freely assigned (and changed) by the designer of the scheme (as long as the port numbers are disjoint at each node). Note, however, that in contrast to the routing case, the actual enumeration of the ports is irrelevant for the correctness of the ancestry scheme (i.e., the fact whether a node is an ancestor of another node has nothing to do with port numbers). Therefore, the only place where the power of designer port model can be exploited by Protocol `Dyn-Anc` is to ease the message delivery protocol, and somehow save on messages. However, by maintaining a table at each node which translates port numbers that are assigned by an adversary to ones assigned by the designer of the scheme, one can operate in the fixed port model and simulate the

scheme in the designer port model using the same messages and labels. Note, this table may increase the memory stored at each node; however, the table is not part of the label, and therefore does not increase the label size. It follows by the above observations that Theorem 1 also holds for the fixed port model.

Our Protocol Dyn-Rout (that operates in the fixed port model) runs Protocol Dyn-Anc. As promised in Theorem 1, for any node  $v$  in the growing tree, given its own ancestry label and the ancestry label of a destination node  $u$ , node  $v$  can find whether  $u$  is its descendant or not. Note that if  $u$  is not a descendant of  $v$  then the desired port number  $\text{Rout}(v, u)$  points at  $v$ 's parent  $p(v)$ . In addition to other components (that will be described soon), the label  $L(v)$  at each node  $v$  contains also the ancestry label  $L_{anc}(v)$  (given by Protocol Dyn-Anc) and the *parent* label  $L_{parent}(v)$ , where the port number  $\text{port}(v, p(v))$  leading from  $v$  to its parent  $p(v)$  is encoded. If  $u$  is not a descendant of  $v$  then the router at  $v$  simply outputs the port number stored in  $L_{parent}(v)$ . However, if  $u$  is a descendant of  $v$  then  $\text{Rout}(v, u)$  leads to one of  $v$ 's children, the one which is an ancestor of  $u$ . The rest of the section is dedicated to show how short labels can be efficiently maintained so that, in the above case, given the labels of  $v$  and  $u$ , node  $v$  will know the port number leading to its child which is an ancestor of  $u$ .

Our general strategy is to adapt the principles of the corresponding (fixed-port) static routing scheme in [5] (the one described in Section 3 in [5]) into the dynamic scenario. In addition to the parent and ancestry labels and other components that will be described later, the label  $L(v)$  of a node  $v$  contains also the *big* label  $L_{big}(v)$  and the *small* label  $L_{small}(v)$ .

The big label at  $v$ ,  $L_{big}(v)$ , is a table containing  $O(\log n / \log \log n)$  tuples of the form  $\langle \text{Port}(v, w), \tilde{L}_{anc}(w) \rangle$ , where  $w$  is a child of  $v$ , and  $\tilde{L}_{anc}(w)$  is some label which informally aims to be the ancestry label of  $w$ . These ‘pointed’ children  $w$  of  $v$  are called *big* and all other children of  $v$  (if any) are called *small*.

Let  $u$  be a descendant of  $v$ . The role of the big label at  $v$  is to allow  $v$  to know whether one of its big children  $w$  is an ancestor of  $u$ , and if so, what is the port number at  $v$  leading to  $w$ . More precisely, given a big child  $w$  of  $v$ , we would first like to determine whether  $u$  is a descendant of  $w$  at a given time, simply by looking at  $L_{anc}(u)$  and the corresponding tuple  $\langle \text{Port}(v, w), \tilde{L}_{anc}(w) \rangle \in L_{big}(v)$ , at the given time. In the static case, this can easily be done by letting  $\tilde{L}_{anc}(w)$  equal  $L_{anc}(w)$  and using the decoder of the ancestry scheme. However, in the dynamic setting, 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 its descendant  $u$  is a descendant of its big child  $w$ , using a rather simple handshake procedure, which is invoked between  $w$  and  $v$  whenever the ancestry label of  $w$  is updated, and which updates  $\tilde{L}_{anc}(w)$  to be the new ancestry label at  $w$ . If  $v$  finds that the destination node  $u$  is a descendant of its big child  $w$ , the router at  $v$  simply outputs the port number  $\text{Port}(v, w)$  (which is encoded in the corresponding tuple in  $L_{big}(v)$ ).

The role of the small labels is to allow  $v$  to determine (in case its descendant  $u$  is not a descendant of any of  $v$ 's big children) which of its ports leads to its small child  $w$  that is an ancestor of  $u$ .

The static scheme of [5] proposed the following. The small label at any big child of  $v$  is the same as the small label at  $v$ . In contrast, the small label at each small child  $w$

of  $v$  is the small label at  $v$  concatenated with  $Port(v, w)$ , the port number at  $v$  leading to  $w$ , i.e.,  $L_{small}(w) = L_{small}(v) \circ Port(v, w)$ . This recursive definition enables  $v$  to extract the port number leading to its small child  $w$  which is an ancestor of  $u$ , simply by comparing the small labels at  $v$  and  $u$ . We refer to the procedure that compares the small labels and extracts the port number  $Port(v, w)$  as Procedure COMPARE&EXTRACT.

In the static case, the  $O(\log^2 n / \log \log n)$  bound on the label size follows from the following easy arguments. First, the ancestry label of each node  $v$  can be encoded using  $O(\log n)$  bits. By the assumption on the size of a port number assigned by the adversary, the parent label of each node is also encoded using  $O(\log n)$  bits. Since each big label contains  $O(\log n / \log \log n)$  tuples, and since each tuple is encoded using  $O(\log n)$  bits, we get that the size of a big label is  $O(\log^2 n / \log \log n)$ . Since each small label is a concatenation of port numbers, the size of a small label is determined by the sum of the sizes of the concatenated port numbers. The main trick in [5] is to choose the big children to be the  $\lambda \sim O(\log n / \log \log n)$  children of  $v$  with maximal weight. This choice of the big children, guarantees that the total number of (concatenated) port numbers in a small label is  $\log_\lambda n = O(\log n / \log \log n)$ . Since each port number is encoded using  $O(\log n)$  bits, the  $O(\log^2 n / \log \log n)$  bound on the label size follows.

We would have liked to adapt that method to the dynamic setting. However, we could not find an efficient way for maintaining the above trick at all times (or even just at quiet times). In particular, it seems that even just to ensure that the weight of the big 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 the child, and the best known scheme that does that [1, 9] already uses  $O(\log^2 n)$  average message complexity. Informally, instead of maintaining the above trick at all times for all nodes, we make sure it holds only locally and only occasionally, whenever we invoke Procedure `Reset`, that is used for balancing and reorganizing certain portions of the tree.

Before dwelling into the description of Procedure `Reset`, let us first describe the update protocol, whose goal is to schedule and trigger the different invocations of Procedure `Reset`.

**The update protocol:** The update protocol operates in *iterations*. A new iteration starts when Procedure `Reset` is invoked on the whole tree. We assume that for every  $j \geq 1$ , during the time period between  $j$ 'th time Procedure `Reset` is completed on the whole tree until the next time such procedure is initiated, each node knows the value  $n_j$ , which is the number of nodes at the beginning of the  $j$ 'th iteration. (In particular, note that  $n_1 = n_0$ .) This assumption is implemented by the `Reset` procedure that is invoked on the whole tree in the beginning of the iteration, as explained later. Let us now describe the operation of the update protocol during the  $j$ 'th iteration, for  $j \geq 1$ .

A node may be either *locked* or *unlocked*. Initially, all nodes are unlocked. When a request (for inserting a child) arrives at an unlocked node, it is handled by Protocol `Dyn-Anc`, that eventually grants it a permit. In contrast, when a request arrives at a locked node  $v$ , it is first put in a queue at  $v$ . When  $v$  becomes unlocked 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 locked node  $v$ , then  $v$  delays the

performance of the corresponding topology change until it becomes unlocked again. In contrast, if  $v$  is unlocked when the permit arrives, then the following happen.

The ancestry label the inserted  $w$  child of  $v$  is given by Protocol Dyn-Anc. The port numbers at  $w$  are given by the adversary (recall, each port number is encoded using  $O(\log n)$  bits). If  $w$  is a leaf, then the port  $Port(v, w)$  is also assigned by the adversary. In this case, the small label at  $w$  is set to be  $L_{small}(v) \circ Port(v, w)$  (thus making  $w$  a small child of  $v$ ). However, if  $w$  is inserted between  $v$  and its child  $u$  then the following happen. First, recall that the port leading from  $v$  to  $w$  remains the same as the port that was leading from  $v$  to  $u$ . We let the small label of  $w$  equal the small of its child  $u$ , that is, if  $u$  was a small child of  $v$  (before  $w$  was inserted) then  $L_{small}(w) = L_{small}(v) \circ Port(v, w)$ , and otherwise, if  $u$  was a big child of  $v$  then  $L_{small}(w) = L_{small}(v)$ . If  $u$  was a big child of its parent  $v$  when  $w$  was inserted then  $w$  becomes a big child of  $v$  (instead of  $u$ ) and the ancestry label in the corresponding tuple is set to be the new ancestry label of  $w$  (this can be implemented instantaneously as the parent  $v$  is the one issuing the ancestry label of  $w$ , when it inserts it). In addition, the big label at  $w$  contains the tuple  $\langle Port(w, u), L_{anc}(u) \rangle$ , thus making  $u$  a big child of  $w$ .

Throughout the execution, each node  $v$  holds a variable  $l(v)$  which informally aims at counting the number of small nodes on the path from  $v$  to the root (including  $v$ , if  $v$  is small). When a node  $u$  is inserted to the tree as a child of its parent  $v$ , its variable  $l(u)$  is set as follows. If  $u$  is a leaf then  $l(u) = l(v) + 1$  (recall, in this case,  $u$  is small). Otherwise, if  $u$  is inserted between  $v$  and its child  $w$  then,  $l(u) = l(w)$  (recall, in this case,  $w$  is a big child of  $u$ , and  $u$  is small iff  $w$  was small before the insertion of  $u$ ). Once  $u$  is inserted to the tree, its variable  $l(u)$  may be updated only by Procedure Reset, in a way to be described soon.

We say that a *triggering event* occurs at some leaf  $u$ , if  $u$  joins the tree with variable that satisfies  $l(u) > 8 \lceil \log n_j / \log \log n_j \rceil$ . This triggering event will result in an invocation of Procedure Reset( $T(\rho)$ ), on some subtree whose root  $\rho = \rho(u)$  is one of  $u$ 's ancestors, called a *anchor* node. The specific choice of the anchor  $\rho$  guarantees that the chosen subtree  $T(\rho)$  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 Reset.) Specifically, the *anchor* node  $\rho$  is the closest ancestor of  $u$  satisfying the following conditions.

*The ANCHOR CONDITIONS:*  $l(\rho) \leq \frac{4 \log n_j}{\log \log n_j}$  and either one of the following holds.

1. a)  $\rho$  is not the root and b)  $\omega(\rho) \leq 2n_j$  and c)  $\rho$  has a small child  $w$  such that  $\omega(w) > \frac{2\omega(\rho) \log \log n_j}{\log n_j}$ ,
2.  $\rho$  is the root of the whole tree  $T$ .

When a triggering event occurs at a leaf  $u$ , Procedure FindAnchor( $u$ ) is invoked at  $u$ . The goal of that procedure is to find the anchor  $\rho(u)$  and to lock the nodes in the subtree  $T(\rho)$  (so that Procedure Reset( $T(\rho)$ ) can subsequently operate on a static tree). Procedure FindAnchor( $u$ ) can be implemented as follows. First, the leaf  $u$  creates an *agent* (see e.g., [9]). This agent then starts walking from  $u$  up the tree while locking 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 node  $v$  for the first time (coming from its child  $w$ ), it initiates a broadcast and upcast operation on the subtree  $T_v \setminus T_w$ . The broadcast locks the nodes in  $T_v \setminus T_w$ , and the upcast lets each node  $z \in T_v \setminus T_w$  know its weight. Subsequently,  $v$  checks whether it satisfies one of the anchor conditions. If it does, then the procedure is completed, and otherwise, the agent continues to  $v$ 's parent. Procedure  $\text{Find\_Anchor}(u)$  incurs  $O(|T(\rho)|)$  messages, where  $|T(\rho)|$  is calculated when the procedure is completed (it will be guaranteed that each such procedure is eventually completed, as explained later on).

In order to avoid collisions between the invocations of different procedures, we do the following. First, if some procedure  $S$  of type  $\text{Find\_Anchor}$  tries to enter a node which initiated a  $\text{Reset}$  procedure that hasn't been completed yet, then Procedure  $S$  first waits for the  $\text{Reset}$  procedure to be completed and only then enters  $v$  and continues its action. Second, regarding collisions between different  $\text{Find\_Anchor}$  procedures, when two (or more) such procedures meet at a node  $v$ , 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 one coming from the child of  $v$  whose corresponding port number at  $v$  is the smallest among all candidates. The formal description of how to implement the above 'traffic rules' is straightforward, and is therefore deferred to the full paper.

Note that the protocol that grants permits to requests is Protocol  $\text{Dyn-Anc}$ . Therefore, it may happen that while some Procedure  $\text{Find\_Anchor}$  tries to 'catch' a subtree and 'lock' it, nodes are continuing to join the subtree and the procedure is never completed. This undesired phenomena can be avoided as follows. Recall that our update protocol operates in iterations. Similarly, also Protocol  $\text{Dyn-Anc}$  works in iterations. Without getting into too much details regarding Protocol  $\text{Dyn-Anc}$ , we just mention the following facts. First, the root decides when an iteration starts. Second, during each iteration the number of nodes in the tree is finite, and third, between iterations, the tree  $T$  remains fixed, i.e., no topology occurs in the tree. Therefore, in order to guarantee that each procedure is completed eventually, when an iteration of Protocol  $\text{Dyn-Anc}$  ends, the next iteration is delayed, and a broadcast and upcast operation is performed on the tree, for making sure that before the next iteration starts, all  $\text{Find\_Anchor}$  procedures are completed. Subsequently, when the upcast is completed, the next iteration of Protocol  $\text{Dyn-Anc}$  can safely start. We are now ready to describe Procedure  $\text{Reset}$ .

**Procedure  $\text{Reset}$ :** As mentioned, for balancing the tree, Protocol  $\text{Dyn-Rout}$  occasionally invokes Procedure  $\text{Reset}$  on different subtrees. (In particular, in the pre-processing stage, before the scenario actually starts, Procedure  $\text{Reset}$  is invoked on the whole initial tree for initializing the labels.) Procedure  $\text{Reset}$ , when invoked on a subtree  $T(\rho)$  reorganizes it and makes it consistent with the whole tree. In the reorganization, the nodes in  $T(\rho)$  are assigned new labels and variables, and may also replace their big children.

It follows from the description of the update protocol that as long as Procedure  $\text{Reset}(T(\rho))$  is operating, all the nodes in  $T(\rho)$  are locked. Thus, the subtree  $T(\rho)$  remains fixed (i.e., no topology change occurs in  $T(\rho)$ ), and moreover, the ancestry label of each node in  $T(\rho)$  remains the same. We assume that when Procedure  $\text{Reset}(T(\rho))$

starts, all the nodes in  $T(\rho)$  are already assigned labels (the initial invocation of Procedure  $\text{Reset}(T)$ , that occurs in the preprocessing stage, is assumed to start with empty labels). The assignment of new labels to the nodes in  $T(\rho)$  is made carefully, in order to keep the scheme correct at all times. For that, we introduce another two labels at each label (that did not exist in the static scheme of [5]) called the *future* label and the *busy* label. The future label at each node  $w$  contains two sublabels, namely, the *future small* sublabel  $fL_{small}(w)$ , and the *future big* sublabel  $fL_{big}(w)$ . The busy label at each node contains a single bit. Initially, all busy labels are zero.

Procedure  $\text{Reset}(T(\rho))$  is initiated at the root  $\rho$  of  $T(\rho)$ . At the first stage, two broadcast and upcast operations are performed on  $T(\rho)$ , which guarantee that upon their completion, the following holds. The future big sublabel at each node  $u \in T(\rho)$  contains the tuple  $\langle port(u, w), L_{anc}(w) \rangle$ , for each child  $w$  of  $u$  that satisfies  $\omega(w) \geq \omega(u) \log \log n_j / \log n_j$ . Each of these children  $w$  of  $u$  is called *future big*, and all other children of  $u$  are called *future small*. The future small sublabel of  $\rho$  is simply a copy of the small label at  $\rho$ ; and the future small sublabel of each  $w \in T(\rho) \setminus \{\rho\}$  is the future small label of its parent  $p(w)$  concatenated with the port number  $Port(p(w), w)$ . i.e.,  $fL_{small}(w) = fL_{small}(p(w)) \circ Port(p(w), w)$ . These two broadcast and upcast operations can trivially be implemented using  $O(|T(\rho)|)$  messages.

Subsequently, after the broadcast and upcast operations are completed, the root  $\rho$  of  $T(\rho)$  creates an agent that performs a DFS tour in the subtree  $T(\rho)$ .

Upon returning to  $w \in T(\rho)$  after visiting all its children, the agent does:

(1) Replaces the small and big labels of  $w$  by the future small and future big labels of  $w$ , respectively, (2) Empties the future labels of  $w$ , and (3) Sets  $w$ 's busy label to 1.

When the agent returns to  $\rho$  after visiting all the nodes in  $T(\rho)$  and implementing the above, it get canceled. Subsequently,  $\rho$  initiates a broadcast and upcast operation for making sure that the following hold. (1) The variable  $l(v)$  at each node  $v \in T(\rho) \setminus \{\rho\}$  is precisely  $l(\rho)$  plus the number of small nodes on the path from  $v$  to  $\rho$  (the path excludes  $\rho$  and includes  $v$ ), (2) The busy label at each node in  $T(\rho)$  is zero, and (3) If  $T(\rho)$  is the whole tree, then each node knows the value  $n_{j+1} = n$  (which is the current tree size).

Finally, when that operation is completed, another broadcast is initiated by  $\rho$  for unlocking all the nodes in  $T(\rho)$ . This completed the description of Procedure  $\text{Reset}(T(\rho))$ . We are now ready to describe the router of the scheme.

**The router:** The goal of the router is to find, given the labels of any pair of nodes  $u$  and  $v$ , the current port number at  $u$  leading to the next node on the shortest path connecting  $u$  and  $v$ , i.e.,  $\text{Rout}(u, v)$ .

Using the ancestry labels and the decoder of Protocol  $\text{Dyn-Anc}$ , 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 the port number stored in its parent label  $L_{parent}(u)$ . If  $v$  is a descendant of one of  $u$ 's big children  $w$  then this is detected using the big label at  $u$  and the ancestry label at  $v$ . In this case, the router outputs  $\text{Rout}(u, v) = Port(u, w)$ , which is encoded in the corresponding tuple in the big label of  $u$ .

If the future label at  $u$  is empty then the router outputs the port number obtained by comparing the small labels of  $u$  and  $v$ , using  $\text{COMPARE\&EXTRACT}$ .

Consider now the case that the future label at  $u$  is not empty. First, if  $v$  is a descendant of one of  $u$ 's future big children  $w$ , then this is detected using the future big

sublabel at  $u$  and the ancestry label at  $v$ . In this case, the router outputs the port number  $Rout(u, v) = Port(u, w)$ , which is encoded in the corresponding tuple in the future big sublabel of  $u$ .

Now consider the case where  $v$  is a descendant of  $u$  but not a descendant of neither one of  $u$ 's big children nor a descendant of one of  $u$ 's future big children. If the future label at  $v$  is not empty then the router outputs the port obtained by comparing the small labels of  $u$  and  $v$ , using COMPARE&EXTRACT; otherwise (the future label at  $v$  is empty), consider two cases. If the busy label at  $v$  is 1, then the router outputs the port obtained by comparing the future small label of  $u$  with the small label of  $v$ , using COMPARE&EXTRACT. Otherwise, if the busy label at  $v$  is 0, then the router outputs the port obtained by comparing the small labels of  $u$  and  $v$ , using COMPARE&EXTRACT.

The proof of the following theorem is deferred to the full version of this paper.

**Theorem 2.** *Consider the fixed port model. Protocol `Dyn-Rout` implements a routing scheme on a growing tree, which is correct at all times. The label size of the scheme is  $\Theta(\frac{\log^2 n}{\log \log n})$  and its average message complexity is  $O(\frac{\log^2 n}{\log^2 \log n})$ .*

The extension of protocol `Dyn-Rout` for supporting also removals of nodes of degree at most 2 is deferred to the full version of this paper.

## 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. I. Abraham, C. Gavoille, D. Malkhi, N. Nisan, and M. Thorup. Compact name-independent routing with minimum stretch. *ACM Transactions on Algorithms* 4(3), (2008).
4. Y. Afek, E. Gafni, and M. Ricklin. Upper and lower bounds for routing schemes in dynamic networks. In *FOCS 1989*, 370–375.
5. P. Fraigniaud and C. Gavoille. Routing in trees. In *ICALP 2001*, 757–772.
6. P. Fraigniaud and C. Gavoille. A Space Lower Bound for Routing in Trees In *STACS 2002*, 65–75.
7. A. Korman. General compact labeling schemes for dynamic trees. *J. Distributed Computing*, 20(3):179–193, 2007.
8. A. Korman. Improved compact routing schemes for dynamic trees. In *PODC 2008*.
9. A. Korman and S. Kutten. Controller and estimator for dynamic networks. In *PODC 2007*.
10. A. Korman and D. Peleg. Compact Separator Decomposition for Dynamic Trees and Applications. *J. Distributed Computing*, 2008, to appear.
11. A. Korman, D. Peleg, and Y. Rodeh. Labeling schemes for dynamic tree networks. *Theory Comput. Syst.*, 37(1):49–75, 2004.
12. A. Korman and D. Peleg. Labeling schemes for weighted dynamic trees. *J. Information and Computation*, 205(12):1721–1740, 2007.
13. A. Korman and D. Peleg. Dynamic routing schemes for graphs with low local density. *ACM Trans. on Algorithms*, to appear. (Preliminary version appeared in *ICALP 2006*).
14. N. Santoro and R. Khatib. Labelling and implicit routing in networks. *The Computer Journal* **28**, (1985), 5–8.
15. M. Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *J. of the ACM* **51**, (2004), 993–1024.
16. M. Thorup and U. Zwick. Compact routing schemes. In *SPAA 2001*, 1–10.
17. J. Van Leeuwen and R. B. Tan. Interval routing. *The Computer Journal* **30**, (1987), 298–307.