

Controller and Estimator for Dynamic Networks

Amos Korman^{*} Shay Kutten[†]

February 20, 2007

Abstract

Awerbuch, Afek, Plotkin, and Saks identified an important fundamental problem inherent to distributed networks, which they called the *Resource Controller problem*. Consider, first, the problem in which one node (called the ‘root’) is required to estimate the number of events that occurred all over the network. The counting can be viewed as a useful variant of the heavily studied and used task of topology update (that deals with collecting *all* remote information). The *Resource Controller problem* generalizes the counting problem further: such remote events are considered as requests, and the counting node, i.e., the ‘root’, also issues *permits* for the requests. That way, the number of request granted can be controlled (bounded).

In the paper by Awerbuch et al., it was assumed that the network is spanned by a tree that may only grow, and only by allowing leaves to join the tree (after receiving a permit). In contrast, the Resource Controller presented here can operate under a more general dynamic model. Specifically, the dynamic model considered in this paper allows both controlled insertions and deletions of leaves as well as controlled insertions and deletions of internal nodes. Despite the more dynamic network model we allow, the message complexity of our controller is always at most the message complexity of the more restricted controller.

All the applications for the previous controller apply also for our controller. Moreover, with the same message complexity, our controller can handle these applications under the more general dynamic model mentioned above. In particular, under the more general dynamic model, the new controller can be transformed into a *Size-Estimation* protocol, i.e., a protocol allowing the root to maintain a constant estimation of the number of nodes in the dynamically changing network using $O(\log^2 n)$ amortized message complexity per topological change (assuming that the number of changes in the network size is “not too small”). The new Size-Estimation protocol can be used to extend many existing labeling schemes supporting different queries (e.g routing, ancestry, etc.) so that these schemes can now operate correctly also under deletion of nodes. These extensions maintain the same asymptotic sizes of the corresponding routing tables and labels of the original schemes and incur only an extra additive cost of $O(\log^2 n)$ to the amortized message complexity. In addition, the new controller can be used to solve the *Dynamic Name Assignment* problem by maintaining unique $\log n + O(1)$ -bit identifiers for the nodes of the dynamic network, using $O(\log^2 n)$ amortized message complexity. Since many static algorithms rely on the fact that the processors have unique and short identifiers, our scheme solving the Dynamic Name Assignment problem may be useful for constructing dynamic variants for these static algorithms.

^{*}Faculty of IE&M, Technion, Haifa 32000, Israel. E-mail: pandit@tx.technion.ac.il. Supported in part by a grant from the Ali Kaufmann Post-Doc fellowship and by a grant from the Israeli Ministry of Science and Technology.

[†]Faculty of IE&M, Technion, Haifa 32000, Israel. E-mail: kutten@tx.technion.ac.il. Supported in part by a grant from the Israel Science Foundation.

1 Introduction

In common sequential settings, an online algorithm (notably, a competitive algorithm, such as those in [4]) must make a decision based on past information, without knowing what the future holds. The main characteristic of a network environment is an additional kind of uncertainty- some nodes may need to make a decision without knowing what already happened in remote locations. The common situation, where both kinds of uncertainties exist, has received very little attention in the literature.

One should stress that the study of each of the above sources of uncertainty separately has been very extensive. In particular, the problem of *updating*- learning what happened in remote places- may be the main type of distributed algorithms actually used in networks. This is because after such information has been learned, the distributed problem is reduced to a better understood sequential one. For example, when a network node has learned the current topology of the network graph, it can compute the best routes to remote nodes by applying (the non-distributed) Dijkstra's shortest path algorithm [6] on the graph represented in the node's own memory. This approach is the one used in the Internet, for example, by the Open Shortest Path First Protocol (OSPF) [7].

This paper addresses problems affected by both kinds of uncertainties. The *token collection* problem [1] is a variant of the updating problem: count (at one center node) the (approximate) number of events that occur all over the network. We solve the above, as well as the more general problem of the *Resource Controller* [14]: in addition to counting a remote event, the algorithm also has to move a *permit* from some center to be consumed at the node requesting the event. More generally, an (M, W) -Controller, must guarantee that the total number of permits granted is at most M . However, if a request is *rejected*, then at least $M - W$ permits are eventually granted (a formal definition appears in Section 2.2).

Given an (M, W) -controller, it is rather easy to use it to derive a protocol solving the token collection problem. However, a controller can be used for additional purposes. For example, a routing scheme may be memory efficient if the size of its distributed data structure, the *routing tables*, is small as a function of the number of nodes. Topology changing operations (say, in an overlay network) may deteriorate the memory efficiency of the routing scheme gradually. The same routing table that was small compared to the size of the initial large network, is now large, compared to the new decreased size of the network. The controller can be used to solve such a problem using low message complexity. The solution runs in iterations as follows. In each iteration: (1) allow (only) roughly $n_0/2$ network changing operations, where n_0 is the initial number of nodes in the iteration, and (2) delay (do not allow) additional operations until every node learns the whole new topology. We describe later several applications where the new controller facilitates efficient solutions for known distributed tasks using the above approach. Additional applications for the controller were presented in [14], where this fundamental problem of a controller was initiated. For example, a controller can maintain short disjoint identifiers at the nodes of the changing network, it can prevent a bug from causing a protocol to send an excessive number of messages, or to use some other resource excessively and unintentionally, etc. See [14] for additional details.

The controller of [14] was designed to work under the *controlled* model, in which the topological changes do not occur spontaneously. Instead, when an entity wishes to cause a topology change at some node u , it enters a *request* at u , and performs the change only after the request is granted a permit from the controller. (This model was rather visionary; today's Peer to Peer applications that did not exist then and, more generally, the now popular overlay networks, come to mind as examples of networks where topological changes can be controlled.) Yet, only changes of one type are allowed in [14]- an insertion of a leaf node. That is, such an insertion is allowed to occur once it receives a permit. No other kinds of topology changes are allowed. Assuming that dynamic model, the message complexity of their controller is $O(N \cdot \log^2 N \cdot \log \frac{M}{W+1})$, where N is the number of nodes ever to exist in the network.

In contrast, the controller and token collectors presented here can be applied for the more general dynamic model where nodes can be inserted (by the environment) also between any two neighbors, and any node (but the root) may also be deleted. In the distributed setting, such an insertion or deletion is

noticed only by the neighbors of the node undergoing the change.

Intuitively, it is not clear how to adapt the previous controller to efficiently handle these additional topology changes. The reason is that the previous controller is based on storing permits at very specific depths of a spanning tree. Specifically, each node has a bin that may store permits. The bins are organized according to an underlying structure called *the bin hierarchy*. Each bin b at a node v has a *level* and a *size* which are determined by the precise distance from v to the root. Bin b also has a *supervisor* bin $sup(b)$ whose location with respect to b also depends on the precise distance from v to the root. A request always walks to a nearby bin b to obtain a permit. If that bin is empty, it replenishes itself from its supervisor bin $sup(b)$ in the bin hierarchy. If $sup(b)$ is also empty then $sup(b)$ tries to replenish itself with permits taken from its supervisor $sup(sup(b))$, etc. It follows that the behavior of a node depends on its precise distance to the root strongly. As mentioned in [14], their controller can operate in the dynamic scenario, assuming that the only topological event that may occur is that a leaf joins the tree. This limited type of a topology update is allowed since it does not affect the depths of the existing nodes and, therefore, does not affect the locations and sizes of the existing bins. However, in the more general dynamic model, a single change in the topology may change many of the above distances, thus spoiling the beautiful combinatorial structure. For example, an insertion of an internal node may move many bins further from the root without them even knowing that fact. It is, therefore, not clear how to adapt their controller so that it can operate efficiently under the more general model.

In contrast to the previous controller, the new (M, W) -Controller is based on different principles. These principles free the controller from depending on precise distances from the root. Hence, the new controller can deal with general insertions and deletions. At the same time, we managed to match the message complexity of the previous controller that was designed for insertions of leaves only. Consequently, we managed to match also the message complexity for the applications mentioned in [14].

1.1 Motivating the model

As in [14], we too assume the *controlled dynamic* model, in which a topological event is delayed until getting a permit to do so from the resource controller. (See the model, Section 2.1.) This model lies between two extremes. On one extreme, lie the static model (used in many studies) as well as the “enough time” model, where the computations that follow one topological change are assumed to occur very fast and are completed before the next topological change occurs (see, e.g. [19, 23, 24]). On the other extreme lies the “chaotic” fully asynchronous and adversarial model. The middle ground controlled model is interesting theoretically, especially since many problems cannot be solved, or can only be solved partially in the “chaotic” model. For example, inserting internal nodes in a rapid succession may prevent any message from reaching its destination, thus making updates impossible.

In addition to the theoretical interest, the controlled model also became a more realistic model in various contexts of overlay networks. Also, it can be used in various environments that provide graceful degradation usually (e.g. using backup methods).

1.2 Additional related work

The controller problem bears similarities to the k -server problem introduced originally in a sequential setting and translated later to the distributed setting [4, 5]. There, *mobile servers* reside in some nodes. When a *request* arrives at some node v , the algorithm must decide which server δ should be moved to node v . Server δ cannot serve a later request that arrives at another node u without first moving from v to u . The cost for a server move from v to u is the distance from v to u . The total cost of an execution (the *move* complexity) is the sum of the costs of the moves. The main difference between the controller problem and the k -server problem is that in the controller problem, multiple “servers” (permits) may be moved together without increasing the cost. In addition, here the “server” is consumed by the request. Finally, here, a request can also be rejected. We treat a reject rather similarly to a permit. That is, a

move of a “reject” to the node with a request is also counted in the move complexity. (However, the number of rejects is not bounded.)

Algorithms that were both competitive and distributed appeared in [8, 9, 2] and in very few later papers. The problem of counting the number of nodes in a growing tree network was suggested (but not solved efficiently) in [11] where an algorithm that created such a growing tree was presented to solve the majority commitment problem in a network where some of the nodes failed *before the algorithm started*. That problem was previously presented in the famous paper of Fischer, Lynch, and Paterson, [10] and solved with $O(n^2)$ messages, each of $O(n \log n)$ bits. An $O(n^2)$ messages (each of $O(\log n)$ bits) solution was given in [12] and an $O(n \log^5 n)$ solution appears in [1]. The granting of permits to requests was studied in [13] in the case that permits are originally distributed in the network nodes according to some probability distribution. The problem of dynamically maintaining routing schemes and other informative labeling scheme representations was studied in [25, 15, 19, 23, 26, 24].

1.3 Our contributions

In this paper, we establish (M, W) -controllers that can operate under the dynamic model allowing both additions and deletions of leaves as well as additions and deletions of internal nodes. Motivated by similarities to the k -server problem, we first present two efficient (M, W) -controllers for the sequential setting. Those can be viewed as a high level description of the distributed controller we present later. The first controller has move complexity $O(n_0 \log^2 n_0 \cdot \log \frac{M}{W+1}) + O(\sum_j \log^2 n_j \cdot \log \frac{M}{W+1})$, where n_j is the number of nodes immediately after the j 'th topological change occurs, and n_0 is the initial number of nodes in the graph. The second controller has move complexity $O(N \cdot \log^2 N \cdot \log \frac{M}{W+1})$, where N is the maximum number of nodes ever to exist simultaneously in the network. We then translate the first sequential controllers to the distributed setting. The message complexity of the resulting distributed controller is asymptotically the same as the move complexity of the sequential one. Let us note that the message complexity of our distributed controller is always at most the message complexity of the more restricted controller in [14] (however, we assume that each message is encoded using $O(\log n)$ bits while in the previous controller each message contains only $O(\log \log n)$ bits; we also use a somewhat larger memory).

In addition, under the more general dynamic model, the new controller can handle all the applications mentioned in [14] using the above mentioned message complexity. In particular, under the more general dynamic model, the new controller solves the Size-Estimation problem using $O(n_0 \log^2 n_0) + O(\sum_j \log^2 n_j)$ messages. The same message complexity is used to solve the Dynamic Name Assignment problem by maintaining disjoint $\log n + O(1)$ -bit identifiers at the nodes of the changing network. Since many static algorithms rely on processors having unique and short identifiers, our solution for the Dynamic Name Assignment problem may be useful for constructing dynamic variants for these static algorithms.

We also use the controller, especially as a size estimator for shrinking trees, for additional applications, not mentioned in [14]. We extend any exact (stretch 1) routing scheme (either dynamic or static and either labeled or name-independent) on trees (e.g., the dynamic schemes in [15, 19, 26] or the static schemes in [16, 28]) to work also under controlled deletions of both leaves and internal nodes. We also extend the dynamic ancestry and routing labeling schemes of [19] and [23] to work also under controlled deletions of internal nodes. In addition, we extend any exact routing scheme (either dynamic or static and either labeled or name-independent) for any type of underlying network to operate also under controlled deletions of degree one vertices. (In the extended routing scheme, if the destination vertex u of a message is deleted, it is not required that the message reaches u). The above mentioned extensions maintain the same asymptotic sizes of routing tables (or labels) of the original schemes (assuming the sizes are measured by reasonable functions¹ of n , the current number of nodes in the graph). The extensions of

¹A reasonable function is a function $f(n)$ satisfying that there exists a constant c , such that for any $n/2 < m < n$, $f(n) \leq c \cdot f(m)$. Note that this condition is satisfied by a function of the form $f(n) = \alpha n^\epsilon \log^\beta n \log^\gamma \log n$, for $\alpha, \epsilon, \beta, \gamma > 0$.

the schemes in [16, 15, 19, 26, 23] can be made with an additive cost of $O(n_0 \log^2 n_0) + O(\sum_j \log^2 n_j)$ beyond the message complexity of the original scheme. The other extensions of the static schemes use additional amortized message complexity of $O(\max\{\frac{\mathcal{M}(\pi, n)}{n} \mid n > 0\})$, where $\mathcal{M}(\pi, n)$ is the maximum message complexity used to assign the routing tables (or labels) on an n -node network.

The paper is organized as follows. The model, the problem and the intuition behind the solutions are presented in Section 2. In Section 3, we describe and analyze the controllers in a sequential form. These description also serve as a high level description of the distributed controller, and expose the main ideas behind it. Section 4 highlights the approach of the distributed implementation, as well as the approach of proving the distributed controller by reducing it to a sequential one. Due to lack of space, the detailed description and analysis of the distributed controller appear in the appendix. The discussion regarding the applications of the new controller is deferred to the the appendix too. Section 5 contains a discussion and open problems.

2 Preliminaries

2.1 The model

Except for the definition of topology changes, we consider the standard point-to-point message passing asynchronous communication network model. The network topology is described by a general undirected graph $\langle V, E \rangle$, where the vertices represent processors and the edges represent bidirectional communication channels operating between neighboring nodes. The messages, which are transmitted over the links of the underlying network, incur an arbitrary but finite delay. Given a tree T (that spans the network graph), and a node $v \in T$, the depth of v is the hop distance between v and the root r of T . The ancestry relation is defined as the transitive closure of the parenthood relation, in particular, a node is its own ancestor. The following types of topological events are considered.

Add-leaf: A new degree one vertex u is *added* as a neighbor of an existing vertex v . The new node u is then considered as a child of v in the spanning tree T .

Remove-leaf: A (non-root) vertex v of degree one is *deleted*.

Add or remove non-tree edge

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 e was a non-tree edge, then we consider this event as composed of adding u as a leaf of v and then adding the non-tree edge (u, w) . Otherwise, if v was w 's parent, then u is a child of v and w is considered a child of u .

Remove internal node: A node u whose degree in T is larger than one is deleted together with all its non-tree edges. The nodes which were u 's children in the tree, become the children of u parent.

A request to delete a node u arrives at u . A request to add a node arrives at the node's parent to be. A request to add a non-tree edge arrives at either one of its endpoints.

As in [14], we assume that a topological event does not occur in a node u until the request is granted a permit to do so from the Resource Controller. When granted the permit, the requesting entity in the requesting node u performs the topological change in a "safe" manner. In the distributed setting, this means that (1) no messages are lost, and (2) any data belonging to the algorithm, stored at a node u that is about to be deleted, is moved to the u 's parent. For more details see the appendix.

A node can distinguish between its own edges. We also assume that at any time during the execution there exists some spanning tree T such that each node knows the port number leading to its parent in T . Different assumptions and methods by which a node can make a distinction between its own ports use different sizes of memory. To make our results applicable for a wide range of models, we assume the relatively wasteful adversarial model [16]. Still, our algorithms memory is rather small (See Section 5). Regarding the message complexity, we assume that each message is encoded using $O(\log n)$ bits, where

n is the number of nodes in the network at the time the message is sent.

2.2 Problem Definition

The input of a controller arrives online in the form of *requests* arriving at different nodes. Initially, an (M, W) -controller has a set of M *permits*, and an infinite set of *rejects* both considered to reside at the tree root. However, permits and rejects can be moved by the controller to other nodes. When a request arrives at a node u , the controller responds eventually by delivering either a permit or a reject to the request. The delivered object (permit or reject) is one of those that originally resided at the root, and is currently residing in u . The delivery of the object to the request consumes the object. A set (even infinite, in the case of rejects) of objects may be moved from a node to one of its neighbors in one message. An (M, W) -controller must satisfy the following correctness conditions.

Correctness conditions

Safety) The total number of requests that were granted permits is at most M .

Liveness) Every request receives either a permit or a reject eventually. If a request is rejected, then the total number of requests that will eventually be granted a permit is at least $M - W$.

Remark: The problem definition in [14] is slightly more general. However, both our algorithm and the algorithm in [14] provide a solution to the more general problem by solving the problem presented above.

2.3 Intuition and high level description

Permits are grouped in ‘packages’ of different sizes which can be moved from one place to another. In contrast to the previous controller, the new controller does not use predetermined locations, and the location of a package has nothing to do with its size. (In fact, a node may store several packages of different sizes).

If a request arrives at a node u having a ‘small’ package, then a permit from that package is granted to the request. Otherwise, u may contain a package that is “too large”, or may not contain a package at all. In either of these cases, an agent is sent up the tree looking for the first package of some size x , located at a distance about x from u . (We use the terms “about” and “roughly” throughout this informal part, in order to expose the intuition better; the exact details of the algorithms appear in the next sections). If no such package exists on the way from u to the root then a package of the appropriate size (roughly the distance between u and the root) is created at the root. When an appropriate package P of size x is found (or created) at a node w (at a distance roughly x from u), its content is distributed along the path from w to u , as follows. This x is roughly 2^i for some i (we call P a level i package). Package P is first moved to a node v at distance roughly 2^{i-1} above u , and then split into two level $i - 1$ packages. One of these packages remains at v and the other, P_2 , is moved to some node z at distance roughly 2^{i-2} above u . P_2 is then split into two packages of level $i - 2$, and the process continues until one level zero package is moved to u . One permit from this level zero package at u is then granted to the request.

To bound the message complexity, we bound the moves of packages. Note, that a permit can be transferred from a package P to a package P' only if P' is about half of the size of P (a result of P 's split). Hence, a permit may belong only to a logarithmic number of packages. Since a move of x permits is to a distance that is about x (about one per permit in the package), the message complexity is low.

The safety condition is satisfied since the root does not issue more than M permits. In order to show that the liveness condition hold, we show that the total number of permits that remain in packages and are not assigned is small. For that purpose, we associate each package P with a set of nodes (some of them may have been deleted already) called the ‘domain’ of P , having the three following properties: (1) The size of P 's domain is about the size of P , (2) the domains of two packages of the same level are disjoint, and (3) the set of existing (non- deleted) nodes in the domain of a package forms a path hanging down from the node holding the package.

The first two properties guarantee that at any time, the number of packages of a given level (and size) is small. Summing over the levels yields an upper bound on the number of permits ‘stuck’ in packages at that time. This yields a lower bound on the number of permits that are eventually granted to requests.

Let us hint how we ensure that domains have the above mentioned properties. Recall, that when the request arrived at u , an agent found the smallest i such that there was a package P of size about 2^i at roughly distance 2^i above u . This means that for $j < i$, there was no packages of size about 2^j at distance about 2^j above u . Hence, there was a path of size roughly 2^j at distance roughly 2^j from u , which is free from a package P' of size 2^j . Some nodes in this path may still belong to a domain of such a level j package P' , even though P' itself does not reside in that path. However, using a counting argument, we manage to locate a subpath of that path that is also of size roughly 2^j , and does not intersect any domain of any existing package of level j . The algorithm recursively moves and splits the package P into smaller level packages. Specifically, a level j package P' is located at the topmost node of the subpath mentioned before, which is considered as the domain of P' .

When topology changes occur, we update the domains without using any communication. Deleted nodes still belong to their domains. If a deleted node w holds one or more packages, they are moved to w 's parent, while retaining their old domains. (Note that w 's children become the children of its parent, therefore, the existing nodes in the domain of the package still have the property that they form a path hanging down from the node holding the package.) If an internal node is inserted between two nodes that belong to the same domain, we associate this new node with that domain. Hence, the existing nodes in that domain still have the property that they form a path hanging down from the node holding the package. To avoid increasing the domain size because of the insertion, we remove the bottom node from the domain. Note that these updates of domains are done only for the sake of the proof, and therefore, the algorithm does not need to notify a node about its domain membership.

3 A Non-Distributed Controller

Motivated by similarities to the k -server problem, we first present an efficient (M, W) -controller for the sequential setting, and then, in the following section, we show how to implement it distributively. The non-distributed controller may also be viewed as a high level description of the distributed one. The move complexity used here (see the discussion of the k -servers in the Introduction) will translate later into the message complexity in the distributed setting.

Note, that the complexity of a trivial controller can be very high. That is, if the only case a permit is moved is directly from the root to the requesting node, the move complexity can reach $\Omega(nM)$, i.e., $\Omega(n)$ per request. On the other hand, if all the requests are known in advance, it is not hard to design an off line sequential algorithm whose move complexity is $O(n)$.

3.1 The algorithm

Let us first assume that there exists a fixed and known upper bound U on the number of nodes n ever to exist in the graph (including the deleted nodes). This assumption is removed later (in Section 3.3).

The algorithm uses a dynamic data structure called *packages*. Each package resides at some node which is referred to as the *host* node of the package. There are two kinds of packages, namely *permit packages* and *reject packages*. Each permit package contains some finite number of permits, and each reject package represents an infinite number of rejects. A permit package may be either *static* or *mobile*. Informally, a static (permit) package is used to grant requests for the node hosting it and a mobile (permit) package is used to deliver sets of permits from place to place. Each permit package (either static or mobile) has a *size*, which is the number of permits in the package. The size of a static package is between 1 and ϕ , where $\phi = \max\{\lfloor \frac{W}{2U} \rfloor, 1\}$; the size of a mobile package is $2^i \phi$ for some integer $i \geq 0$. Consider a mobile package of size $2^i \phi$. For convenience, we call i the *level* of the package.

Initially, there are no packages anywhere. The following actions are supported by the data structure:

- (1) The creation of a package P residing at the root. A mobile package is created together with a level and a size, which is determined according to the level. This operation increments a variable called **Issued** by the size of P . The variable **Issued** is kept at the root and is initially set to zero. If a reject package is created, then it represents infinitely many rejects.
- (2) The *split* of a package into two packages: when a mobile package of level $i > 1$ splits, both resulting packages are mobile packages of level $i - 1$. When a mobile package of level 1 splits, one of the resulting package is a mobile 0-level package and the other is a static package containing ϕ permits. When a reject package splits, it splits into two reject packages (each representing infinitely many rejects).
- (3) The *move* of a package from its location node to a new location node.
- (4) The *granting* (respectively, delivering) of a permit (resp. reject) from a static (resp. reject) package in a node u to a request in the same node. The granting of a permit decreases the size of the static package by one. If, consequently, the size of the static package becomes zero then the package is canceled, i.e., it no longer exists in the data structure.

We need the following definitions. Let $\psi = 4 \log(U + 2) \cdot \max\{\lceil \frac{U}{W} \rceil, 1\}$. Given a node u and a given time t , a *filler* node w with respect to u is a node w satisfying the following two conditions at time t :

- a) w contains a mobile package P of level j .
- b) if $j = 0$ then $0 \leq d(u, w) \leq 2\psi$, otherwise $2^j < d(u, w) \leq 2^{j+1}\psi$.

We are now ready to describe Protocol GRANTORREJECT(u) which is applied by the algorithm in response to an arrival of a request at some node u .

Protocol GRANTORREJECT(u)

1. If there exists a reject package at u then the request is rejected. Otherwise the following happens.
2. If there exists a static package P of size $S > 0$ residing at node u , then a single permit in P is granted to the request. Subsequently, the size of P is reduced to $S - 1$. If, consequently, the size of P becomes zero then P is canceled. If the request is for a topological event τ then consider the following cases.
 - If τ is of type remove leaf or remove internal node and u , the vertex to be removed, holds several packages, then these packages are moved to u 's parent. Subsequently, u is removed.
 - Otherwise, the requested event takes place when the request is granted the permit.

If there is no static package at u then the following happens.

3. If there exists at that time an ancestor of u that is a filler node with respect to u , then let $\rho(u)$ be such a filler node that is the closest to u . Also, let $P(u)$ and $j(u)$ be such that $\rho(u)$ has the package $P(u)$ of level $j(u)$ satisfying the conditions mentioned above, in the definition of a filler node.

Otherwise (no filler node exists), let $j(u) \geq$ be the smallest integer such that $d(u, r) \leq 2^{j(u)+1}\psi$.

In this case, a mobile package $P(u)$ of level $j(u)$ is created at the root r . Recall, that the creation of $P(u)$ increments the variable **Issued** by $2^{j(u)}\phi$. If, subsequently, **Issued** $> M$, then the request is rejected. In addition, a reject package is placed in every node. This is done by first creating a reject package at the root, and then using splitting and standard broadcast operations.

If the package was not rejected, then the handling of the request proceeds as follows.

4. For each $k \in \{0, 1, 2, \dots, j(u) - 1\}$, let u_k be the ancestor of u satisfying $d(u, u_k) = 3 \cdot 2^{k-1}\psi$. Apply the following procedure PROC(P) recursively with $P = P(u)$.
PROC(P): Given a package P of level k at a vertex w , act as follows.

- If $k > 0$ then move P from w to vertex u_k . Then, split P into two packages P_1 and P_2 , each of level $k - 1$. Leave P_1 at u_k and apply the algorithm recursively for P_2 .
- If $k = 0$ (including the case where $j(u) = 0$) then package P is moved to u and becomes static. The request is then granted to u from P according to item 2 above.

3.2 Correctness and Complexity

Given a time t , an *existing* package (respectively, node) is a package (resp., node) that exists in the graph at time t . For analysis purposes, every existing mobile package P is associated with a set of (not necessarily existing) nodes, called the *domain* of package P . The following invariants are maintained:

The domain invariants

- 1) For every k , the domain of each existing mobile package of level k contains $2^{k-1}\psi$ nodes.
- 2) For every k , the domains of the existing mobile packages of level k are disjoint.
- 3) The existing nodes (if such exist) in the domain of every mobile package form a subpath hanging down (away from the root) from some child of the node holding the package.

Let us now define the domains and show that the domain invariants indeed hold. Initially, there are no packages and no domains. When a package is cancelled or becomes static, its domain is cancelled. Similarly, when a package splits, its domain is cancelled and new domains should be given to the new packages resulted from the split. A package is *formed* at some time t if, at time t , the package is either created (at the root) or is resulted from a split. We first define the domain of a newly formed mobile package. Note that a package may be formed only after a request arrives at some node u .

Case 1) If the filler node $\rho(u)$ exists (item 3 of the Protocol) and $j(u) = 0$ then no package is formed.

Case 2) If $\rho(u)$ does not exist and $d(u, r) \leq 2\psi$ then no new mobile package results (Indeed, a level zero mobile package is created at the root, but it is then moved to u immediately and becomes static.)

Case 3) If $\rho(u)$ exists and $j(u) > 0$ then let $P(u)$ be the level $j(u)$ package residing at $\rho(u)$. After the recursive procedure $\text{PROC}(P(u))$ is completed, we have the following. For $k \in \{0, 1, 2, \dots, j(u) - 1\}$, one level k mobile package P_k is located at u_k above u , such that $d(u, u_k) = 3 \cdot 2^{k-1}\psi$. For every k , the domain $\text{Dom}(P_k)$ associated with the package P_k is the set of vertices w on the path connecting u and u_k which satisfy $1 \leq d(w, u_k) \leq 2^{k-1}\psi$. As shown below, no node in $\text{Dom}(P_k)$ belongs then to another domain of the same level.

Case 4) If $\rho(u)$ does not exist and $d(u, r) > 2\psi$ then let $P(u)$ be the package created by the root. Recall, that Procedure $\text{PROC}(P(u))$ is applied. The domains of the newly formed mobile packages resulting from the recursive application of Procedure $\text{PROC}(P(u))$ are defined as in the previous case (assuming $\rho(u)$ is the root and that package $P(u)$ resides at $\rho(u)$). Note that we do not need to define a domain for $P(u)$ since it is split immediately after being created.

Let us now define how the domain of an existing mobile package P may be affected by a topological event τ occurring in the (existing) nodes in $\text{Dom}(P)$.

Case 5) An event of type add leaf or of type add or remove non-tree edge, has no affect on $\text{Dom}(P)$.

Case 6) If τ is of type add internal node, and u , the added vertex, belongs to $\text{Dom}(P)$, then u is added to domain $\text{Dom}(P)$ and the bottom most existing node in $\text{Dom}(P)$ is removed from P 's domain.

Case 7) If τ is of type remove leaf or remove internal node, and u , the removed node, belongs to $\text{Dom}(P)$, then u continues to belong to $\text{Dom}(P)$, but as a deleted node and not as an existing one.

Claim 3.1 *The domain invariants hold at all times.*

Proof: Clearly, the invariants hold initially, when there are no packages. Assume by induction, that they hold just before the next time t where either a package is formed or a topological event occurs.

First, consider the case that at time t , a package is formed. New domains are defined only due to an application of $\text{PROC}(P(u))$. After Procedure $\text{PROC}(P(u))$ is completed, for every $k \in \{0, 1, 2, \dots, j(u) -$

1}, one level k new mobile package P_k is located at vertex u_k above u , such that $d(u, u_k) = 3 \cdot 2^{k-1}\psi$. The first and third domain invariants follow directly from the description in Cases 3 and 4 in the definition of the domains above. It is left to show that the second domain invariant holds.

Fix some $k \in \{0, 1, 2, \dots, j(u) - 1\}$. Let I_k denote that subpath containing the ancestors w of u satisfying $2^k\psi < d(u, w) \leq 2^{k+1}\psi$. Note, that the definition of $j(u)$ implies that before $\text{PROC}(P(u))$ is applied, there is no mobile package of level k in subpath I_k . Hence, there is no mobile package of level k in any vertex in $\text{Dom}(P_k)$, since $\text{Dom}(P_k) \subset I_k$. Therefore, by the third domain invariant, $\text{Dom}(P_k)$ does not intersect with any other domain of a package of level k residing at a *descendant* of u_k . Moreover, by the first and third domain invariants, and by the fact that $I_k \setminus \text{Dom}(P_k)$ does not contain any mobile package of level k either, we obtain that $\text{Dom}(P_k)$ does not intersect any domain of any other level k package residing at an *ancestor* of u_k . Therefore, the second domain invariant also holds at time t .

Now consider the second case where domains may change at time t , that is, a topological event τ occurs. In case 5 above, neither a domain nor a package changes, hence, the domain invariant continues to hold. The same applies if the topological event concerns nodes which are not in any domain and do not hold any package. Now, assume that τ is of type add internal node, and vertex u is added at time t between two existing vertices v and w . Case 6 above is applied. P 's domain loses its bottom most node but gains node u that is new, and hence has not belonged to any domain in the level of P . Therefore, the first and second domain invariants continue to hold. In addition, the removal of the node from the domain does not disconnect the path which is $\text{Dom}(P)$ since the removed node is the bottom most. Similarly, the addition of the new node u keeps $\text{Dom}(P)$ as a path since u (and edges (v, u) and (u, w)) replace edge (v, w) on the path. If τ is of type remove leaf or remove internal node, and just before time t , the removed node u belonged to $\text{Dom}(P)$ for some package P , then u continues to belong to $\text{Dom}(P)$. Therefore, the first domain invariant holds. In addition, no node was added to any domain, therefore, the second domain invariant still holds as well. If, just before time t , the removed node u did not have any package, then clearly, the third domain invariant holds as well. If the removed node u contained several packages, then they were moved at time t to u 's parent, and since u 's children become the children of its parent, then the third domain invariant holds also at time t . This completes the proof. \blacksquare

3.2.1 Correctness

Lemma 3.2 *The correctness conditions hold for the non-distributed controller.*

Proof: The safety condition for the number of grants issued is clearly maintained, by item 3 in the description of the algorithm. In addition, it is easy to show that every request granted corresponds to a permit issued. Hence, the safety condition holds for the requests granted too.

Now, consider the first time a request is rejected. Let us first bound the sum of the sizes of the currently existing mobile packages. By the first two domain invariants, the number of level k mobile packages is at most $\frac{U}{2^{k-1}\psi}$. Note, that $\frac{\max\{W/2U, 1\}}{\max\{U/W, 1\}} \leq \frac{W}{U}$. Therefore, the sum of the sizes of the level k mobile packages is at most,

$$\frac{2^k U}{2^{k-1}} \cdot \frac{\phi}{\psi} \leq \frac{2^k U}{2^{k-1}} \cdot \frac{W}{4U \log(U+2)} = \frac{W}{2 \log(U+2)}.$$

By the first domain invariant, the domain of a level k package is $2^{k-1}\psi$ and, therefore, $2^{k-1} \leq U$. It follows that $k \leq \log U + 1$, hence, the number of levels is at most $\log U + 2$. It follows that the sum of the sizes of all the mobile packages is at most $W/2$.

Let us now bound the sum of the sizes of the existing static packages. If $W < 2U$ then $\phi = 1$, hence, there are no static packages (once a static package of size 1 is formed, the single permit in it is immediately granted and the package is canceled immediately). If, on the other hand, $W \geq 2U$, then $\phi \leq W/2U$. Therefore, the sum of the sizes of the static packages is, at most, $U \frac{W}{2U} = W/2$.

It follows that the sum of the sizes of the all the packages (both mobile and static) is at most W . Therefore, at any given times, the total number of permits in all the currently existing packages is at most W . Note, that if a request is rejected then at least M permits were issued by the root. It follows that at least $M - W$ requests were granted to requests. This proves the lemma. ■

3.2.2 Complexity

Lemma 3.3 *The move complexity of the non-distributed algorithm is $O(U \frac{M}{W} \log^2 U)$.*

Proof: Permits and rejects move only in packages. Clearly, the move complexity for all reject packages is at most U . A package may be moved up the tree only as a result of a deletion. Specifically, if a node u holds several packages and is given a permit to delete itself, then, as explained in item 2 of the description of the algorithm, it first moves its packages to its parent. Therefore, since the number of deletions is at most U , the total move complexity resulted from such moves is U .

The only other types of moves of permit packages are as a result of applying procedure $\text{PROC}(P)$. Therefore, throughout the scenario, each mobile package P moves at most twice. Once when P is created during the application of some procedure $\text{PROC}(P')$ and once if P is the level k package $P(u)$ found at the filler node $\rho(u)$ and moves to u_k (if $k > 0$) or to u (if $k = 0$). Both these moves are to distance $O(2^k \psi)$ where k is the level of P . Since at most M permits are issued, and since a permit may belong to at most one level k package, the total number of packages of level k ever to exist is $\frac{M}{2^k \phi}$. Therefore, the sum of the costs of the moves made by packages of level k is $O(2^k \psi \frac{M}{2^k \phi}) = O(M \frac{\psi}{\phi}) = O(U \frac{M}{W} \log U)$. Since there are $O(\log U)$ levels k , the lemma follows. ■

The move complexity can be further reduced, as in Section 6 of [14]. In order to deal with the cases where M/W is large, one can iterate the controller $O(\log \frac{M}{W+1})$ times. In each iteration, the ‘waste’ is at least halved. First set $M_0 = M$. In the i 'th iteration the controller is initiated with the parameters $(M_i, M_i/2)$. When the i 'th iteration terminates, the algorithm counts the number L of unused permits in the packages that exist at that time. Then, instead of rejecting a request, the algorithm sets $M_{i+1} \leftarrow L$ and the $i + 1$ 'st iteration starts. After $i' = O(\log \frac{M}{W+1})$ iterations, $M_{i'+1}$, the number of unused resources in the existing packages is within a constant multiplicative factor of W and the $i' + 1$ iteration is initiated with parameters $(M_{i'+1}, W)$. This leads to the following lemma.

Lemma 3.4 *The move complexity of the non-distributed algorithm is $O(U \cdot \log^2 U \cdot \log \frac{M}{W+1})$.*

3.3 The case that no fixed U is known

We now handle the general case, where we do *not* assume that we know in advance a fixed upper bound U on the number of nodes ever existing in the graph. In the proof of the following theorem, a controller for the general case is constructed by running the above algorithm in iterations. Since this idea is similar to the one described in Section 5 of [14], we defer the proof of the following theorem to the appendix.

Theorem 3.5 • *There exists a non-distributed controller whose move complexity is $O(n_0 \log^2 n_0 \cdot \log \frac{M}{W+1}) + O(\sum_j \log^2 n_j \cdot \log \frac{M}{W+1})$, where n_j is the number of nodes immediately after the j 'th topological change occurs.*

- *There exists a non-distributed controller whose move complexity is $O(N \cdot \log^2 N \cdot \log \frac{M}{W+1})$ where N be an upper bound on the number of nodes. (It is not required that N is known in advance).*

4 An Overview of the Distributed Implementation

The details and the analysis of the distributed implementation of the controller appear in the appendix because of the lack of space. We note that the non-distributed controller was constructed in such a way that it can be implemented distributively. Moreover, the proof of the distributed version is by a reduction to the non-distributed one. Let us give here an overview.

The arrival of a request at a node u creates a mobile agent at u . If there is no static package at u then the agent climbs the tree (carried by messages) until it reaches either a filler node or the root. It then takes the package located there and performs $\text{PROC}(P)$ by walking down the tree.

The only real difference from the sequential one is the fact that an agent cannot act instantaneously on multiple nodes, while in the non-sequential algorithm, each request is handled fully before the next request arrives. To ease the proof, the instantaneous action is simulated using locks, and using the assumption that topology changes occur in a “safe” manner (see Section 2.1). An agent performs operations on the data structure only after it has locked all the nodes it needs to touch.

Finally, the distributed implementation is proved by mapping each distributed execution to an execution of the non-distributed controller. In the simulation, a request that arrives at some time t_1 but is granted later at time $t_2 > t_1$ in the distributed execution, is mapped into a request that arrives and is granted at time t_2 in the non-distributed execution. The following theorem then follows directly from the analysis of the non-distributed controller.

Theorem 4.1 *There exists a distributed (M, W) -controller whose message complexity is $O(n_0 \log^2 n_0 \cdot \log \frac{M}{W+1}) + O(\sum_j \log^2 n_j \cdot \log \frac{M}{W+1})$, where n_j is the number of nodes immediately after the j 'th topological change occurs, and n_0 is the initial number of nodes in the graph.*

5 Conclusion

The memory complexity of the distributed algorithm depends on the model to some degree. If a large number of requests can be injected by the environment at once, a node may need to use memory to hold all these requests. To account only for memory used by the algorithm, we prefer to assume that a node can refuse to accept an additional request until it finishes handling the current one. In this case it is possible to bound the memory to be logarithmic per edge. Further reduction may be possible, depending on the model of the ports (see a discussion in the appendix).

The locking of nodes can increase the time complexity. However, if nodes are not locked, it seems hard to ensure a small message complexity in the face of the insertion of a large number of internal nodes. Another reason for locking is the saving of memory. We note that this is also one of the reasons nodes are locked in [14] in one of their controllers (the other reason is to save in message size).

The message size used in this paper is $O(\log n)$, while the (locking version) controller of [14] uses messages of size $O(\log \log n)$. Intuitively, in the new controller, an agent needs to count the number of steps it moves. Such counting was not necessary in the controller of [14] due to the fixed locations that it used for bins. Instead of counting steps, a message just walked up until it reaches the supervisor bin. An open problem is, can this be matched in a dynamic network, where the locations cannot be fixed?

Another interesting question is whether the message complexity of the controller can be reduced. Put differently, can the competitive ratio be reduced? Finally, we have shown that it is possible to match the message complexity of the previous controller in the more general setting. It would be interesting to find out whether for optimal controllers there exist inherent gaps in the complexities of controllers for more limited dynamic models and the complexities of controllers for more general dynamic cases.

References

- [1] Y. Afek and M. E. Saks. Detecting Global Termination Conditions in the Face of Uncertainty. PODC 1987, pp. 109-124.
- [2] B. Awerbuch, Y. Bartal, and A. Fiat. Competitive distributed file allocation. STOC 1993, pp. 164-173.
- [3] L. Barrire, P. Flocchini, P. Fraigniaud, and N. Santoro. Can we elect if we cannot compare? SPAA 2003, pp. 324-332.
- [4] M.S. Manasse, L.A. McGeoch, and D.D. Sleator. Competitive Algorithms for On-Line Problems. Proceedings of the 20th Ann. ACM Symp. on Theory of Computing, May 1988, pp. 322-333.
- [5] Y. Bartal and A. Rosen. The Distributed k -Server Problem- A Competitive Distributed Translator for k -Server Algorithms. The 33rd FOCS pp. 344-354, 1992.
- [6] E. W. Dijkstra. A note on two problems in connexion with graphs. Numerische Math. 1 (1959), S. pp. 269-271.
- [7] J. Moy - 1994 - RFC 2328, April 1998.
- [8] B. Awerbuch, S. Kutten, and D. Peleg. Competitive Distributed Job Scheduling. STOC 1992, pp. 571-580.
- [9] Y. Bartal, A. Fiat, and Y. Rabani. Competitive Algorithms for Distributed data Management. J. Comput. Syst. Sci. 51(3): 341-358 (1995).
- [10] M. J. Fischer, N. A. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. J. ACM 32(2), pp. 374-382 (1985).
- [11] S. Kutten. Optimal Fault-Tolerant Distributed Construction of a Spanning Forest. Information Processing Letters, Vol. 27, pp. 299-307, May 1988.
- [12] R. Bar-Yehuda and S. Kutten. Fault-Tolerant Majority Commitment. J. of Alg. Vol. 9, pp. 568-582, 1988.
- [13] N.A. Lynch, N.D. Griffeth, M.J. Fischer and L.J. uibas. Probabilistic Analysis of a Network Resource Allocation Algorithm. Inf. Cont. 68, 47-85.
- [14] Y. Afek, B. Awerbuch, S.A. Plotkin and M. Saks. Local management of a global resource in a communication network. J. ACM 43, (1996), 1-19.
- [15] Y. Afek, E. Gafni, and M. Ricklin. Upper and lower bounds for routing schemes in dynamic networks. In Proc. 30th Symp. on Foundations of Computer Science, 1989, 370-375.
- [16] P. Fraigniaud and C. Gavoille. Routing in trees. ICALP 2001, pp. 757-772.
- [17] C. Gavoille, D. Peleg, S. Pérennes and R. Raz. Distance labeling in graphs. In Proc. 12th ACM-SIAM Symp. on Discrete Algorithms, pages 210-219, Jan. 2001.
- [18] S. Kannan, M. Naor, and S. Rudich. Implicit representation of graphs. In SIAM J. on Discrete Math 5, (1992), 596-603.
- [19] A. Korman. General Compact Labeling Schemes for Dynamic Trees. In Proc. 19th International Symposium on Distributed Computing, Sep. 2005.
- [20] M. Katz, N.A. Katz, A. Korman and D. Peleg. Labeling schemes for flow and connectivity. SIAM Journal on Computing 34 (2004), 23-40.
- [21] A. Korman and S. Kutten. Distributed Verification of Minimum Spanning Trees. PODC 2006.
- [22] A. Korman. Labeling Schemes for Vertex Connectivity. *Submitted*.
- [23] A. Korman, D. Peleg, and Y. Rodeh. Labeling schemes for dynamic tree networks. ToCS 37 (2004), pp. 49-75.
- [24] A. Korman and D. Peleg. Labeling Schemes for Weighted Dynamic Trees. ICALP 2003, pp. 369-383.
- [25] A. Korman and D. Peleg. Dynamic Routing Schemes for General Graphs. ICALP 2006.
- [26] A. Korman and D. Peleg. Compact Separator Decomposition for Dynamic Trees. *Submitted*.
- [27] E. Korach, S. Kutten, and S. Moran. A Modular Technique for the Design of Efficient Distributed Leader Finding Algorithms. ACM TOPLAS Vol. 12, No. 1, pp. 84-101, 1990.
- [28] M. Thorup and U. Zwick. Compact routing schemes. SPAA 2001, pp. 1-10.

Appendix

A Proof of Theorem 3.5

Proof: Let us first construct a controller with the complexity stated in the first part. In the i 'th iteration, we assume that $U = U_i$, and execute the (M_i, W) -controller, where U_i and M_i are defined below. The first iteration is initiated with $M_1 = M$, and with $U_1 = 2n_0$, where n_0 is the initial number of nodes in the graph. Let N_i denote the number of nodes in the graph at the beginning of the i 'th iteration and let Y_i denote the number of (granted) requests during the i 'th iteration. Let Z_i be the number of *topological changes* that occur during the i 'th iteration. The i 'th iteration is terminated when Z_i reaches $N_i/2$. Note that N_i , Z_i and Y_i (which may be different than **Issued**) can be computed easily in the sequential setting. (Later, in the distributed setting, we shall use a second controller to estimate Z_i , and count the exact Y_i and N_{i+1} at the end of each iteration.) When the i 'th iteration terminates, we first initialize the data structure by removing all the existing packages from the graph, and by setting **Issued** = 0. The $i + 1$ 'st iteration is then initiated with $U_{i+1} = 2N_{i+1}$, $M_{i+1} = M_i - Y_i$. Clearly, U_i satisfies $U_i/4 \leq n \leq U_i$ during the i 'th iteration.

The fact that the modified controller is correct is clear. For each iteration i , let j_i be the number of topological changes that occurred until the i 'th iteration was terminated. For $i > 1$, we have $U_i = \Theta(N_i) = \Theta(j_i - j_{i-1})$. By Lemma 3.4, the move complexity during the i 'th iteration is $O(U_i \log(\frac{M_i}{W+1}) \log^2 U_i)$. Therefore, for $i > 1$, the move complexity during the i 'th iteration is

$$O((j_i - j_{i-1}) \cdot (\log^2 U_i \cdot \log \frac{M_i}{W+1})) = O\left(\sum_{j=j_{i-1}+1}^{j_i} \log^2 n_j \cdot \log \frac{M}{W+1}\right).$$

Therefore, the total move complexity is $O(n_0 \log^2 n_0 \cdot \log \frac{M}{W+1}) + O(\sum_{i>1} \sum_{j=j_{i-1}+1}^{j_i} \log^2 n_j \cdot \log \frac{M}{W+1}) = O(n_0 \log^2 n_0 \cdot \log \frac{M}{W+1}) + O(\sum_j \log^2 n_j \cdot \log \frac{M}{W+1})$.

To construct the controller with the complexity stated in the second part of the theorem, we change the definition of an iteration as follows. An iteration is terminated (and a new iteration starts) only when the number of nodes is doubled from the beginning of the iteration. The analysis is very similar to the one above. ■

B Distributed Implementation: sketch

Before describing the distributed implementation of the sequential algorithm, let us first discuss some issues regarding the removal of nodes in the distributed setting. As in the sequential setting, when the permit to delete a node u is granted to u , the requesting entity in the requesting node can then perform the topological change. However, in the distributed setting, we assume that the deletion is made in a “safe” manner. In particular, (1) no messages are lost. (A message sent to a parent who is being deleted is either actually both sent to and arrive at the parent before it is deleted, or is sent to and arrive at the new parent). (2) in the case of a node's deletion, any data belonging to the algorithm, stored at the node, is moved to the node's parent. (If the parent is also deleted subsequently, its removal does not take place before this data is further moved to its own parent, etc.)

Note, that ensuring the above “safe” manner 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 deleting some node 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. As another example, any addition of an edge may require a handshake between the endpoints of the edge (this, for example, is the

case when the edge represents a TCP connection in an overlay network). Such additional protocols may be interesting in themselves. However, they are beyond the scope of the current paper.

Another issue regarding the deletion of a node is whether there are additional unfulfilled requests that arrived at that node from the environment, and are now waiting at the node for either a permit or a reject. Recall (from Section 5), that we do not consider the memory space these requests may occupy as a part of the space of the algorithm. Still, it may be possible for the algorithm to treat some requests of this kind. This is not necessarily possible (or even desirable) in all the cases. That is, some such requests may lose their meaning if the node is deleted, and the environment may no longer be interested in having them granted permits. (Examples are additional requests to delete u after it is already deleted.) There may be cases that some such requests are still meaningful and can be treated even if the node is deleted. In such cases, the “safe” manner assumption includes taking care of such requests. Then, the “safe” manner includes identifying such requests and moving the requests too to the parent node. Again, the way the deletion requesting entity handles this is beyond the scope of this paper.

We note that, while the requirement for the “safe” manner may delay the requesting entity in the actual deletion, we no longer count the node in the number of existing ones. For example, no request arrives at such node, a routing algorithm is not required to be able to locate it, etc.

We now begin the description of the distributed controller. As in the sequential setting, we first assume that a fixed upper bound U on the number of vertices ever existing in the network (including the deleted vertices) is given in advance. The removal of this assumption is discussed in Subsection B.4.

For simplicity of presentation, we describe the distributed implementation using more than one protocol layers. To simplify the higher layer, the algorithm is written as an algorithm for a *mobile agents*. By doing that, we manage to push most of the less interesting details into the lower layer that supplies services to the agent, rather than to the actions of the agent.

B.1 High Layer Distributed Implementation

A mobile agent is a process that can move from a node to another node by issuing a move instruction. This instruction moves the agent process together with its state, including its program counter and its variables. Hence, the agent can read and write its own variables even after the move. While at a node, the agent can access information stored at the node. (This storage is called a *whiteboard* e.g. in [3]). The whiteboard is not moved when the agent moves. It remains at the node and can be read and written by any agent that visits it, but only while visiting the node. (Of course, an agent can copy the content of the whiteboard to a variable it carries with itself, but this may increase the communication complexity, since additional information is then carried.) To reduce issues related to concurrency within one node, only one agent is active at a node at one time. That is, the agent handles an event (e.g. the arrival at the node) atomically. After that, the agent either leaves the node, or waits, while other agents may arrive and handle their events. Such algorithms are described, for example, in [27, 3]. For additional definitions and implementation details see [27].

The arrival of a request at a node u (as an input from the environment) creates an agent at u . (Recall, that to account only for the memory used by the controller, we assume that the next request may arrive only after the agent of the current request terminates.) This agent starts traveling in the tree and after some time returns to u with the answer for the request (either a grant, or a reject). We assume that when the agent returns to u , the agent knows how to communicate with the entity in u 's environment that issued the request. This communication is needed in order to deliver the answer. This communication is beyond the scope of the current paper. (For example, the requests may be injected to u by the environment to a queue, and the request handled by the agent is the one at the top of the queue.)

As in the sequential algorithm, there are two kinds of packages, namely *permit packages* and *reject packages*. Each permit package contains some finite number of permits, and each reject package represents

an infinite number of rejects. A permit package may be either *static* or *mobile*. The whiteboard at a vertex w contains the set of the packages residing at w . For each such permit package P , the whiteboard contains a) a flag indicating whether P is static or mobile, and b) the size S of P (note, that the level of P can be calculated from its size). In addition, the whiteboard at w contains the boolean variable *state* which may receive either the value ‘locked’ or the value ‘unlocked’. A node whose state variable is ‘locked’ (respectively, ‘unlocked’) is referred to as *locked* (resp., *unlocked*). The whiteboard at the root also contains the variable **Issued**, initially set to zero, and the parameter M .

The agent algorithm uses a *taxi* algorithm as a carrier [27]. That is, in every node, the agent algorithm has an interface via which it can issue instructions to the taxi algorithm. One such instruction is to move towards the root. If the agent issues this *Up* instruction while it is in some node w that is not the root, the agent process is suspended and then awakened in the parent of w . If the agent created at u is at some ancestor w of u , a second kind of instruction it may issue to the taxi algorithm is *Down*. When the agent at w issues this *Down* instruction, its process is suspended and then awakened in the child of w on the path connecting w and u . In addition, the agent at node w may query the taxi algorithm for the distance between u and w . This is done by the command *Distance*. Another query the agent can issue is *DistToTop* which returns the distance to the topmost node (closest to the root) ever reached by this agent.

B.1.1 The algorithm

1. The arrival of a request at a node u (as an input from the environment) creates an agent at u to which we refer as the agent of u , though a unique identity for u is not necessary. The agent is created with one variable: *Bag* (initially empty). If the agent is created at an *unlocked* node, it immediately locks it, by writing ‘locked’ in the state variable in the node’s whiteboard.

- (a) If an agent is created at a node which was already locked by some other agent or reaches such a node at some later point, then the following happens.

The agent waits passively in the locked node until the node is unlocked (by some other agent). If more than one agent is waiting for a node to become unlocked, then the agent that resumes execution (when the node becomes unlocked) is the first one waiting, according to the First In First Out discipline. (We assume that local computation take zero time, hence, once the node becomes unlocked, the first agent waiting at the node is dequeued before any new agent reaches the node from another node). When an agent is dequeued from a node’s queue it continues to behave as if it has just entered the node.

- (b) A node that contains a reject package is called a *reject* node. If the agent is created at a reject node then a reject is delivered to the request. Otherwise, if the agent of u starts moving and then reaches a reject node (for the first time), then the agent walks to its origin node u , delivers a reject, then walks to the topmost node it ever reached, and then walks down to u , unlocking every node it locked (including u). (To arrive to its topmost node v , the agent issues the command *Up* to the taxi algorithm *DistToTop* times.)

Otherwise, if u is unlocked and does not contain a reject package, then the agent acts according to the following items.

2. If the agent is still at u (i.e, the vertex where the agent was created) and u ’s whiteboard contains a static package P of size S , then the following happens. A single permit in P is granted to the request as an output to the environment. Subsequently, after the request is granted, the agent rewrites the size of P in u ’s whiteboard to be $S - 1$. If, consequently, the size of P is zero then the agent cancels P . Canceling means erasing P from u ’s whiteboard, together with all of P ’s attributes (i.e., P ’s size and corresponding flag).

If the request was to perform a topological event τ then the following happens.

- (a) If τ is of type remove leaf or remove internal node and u , the vertex to be removed, holds several packages or a non-empty queue (storing agents), then these packages and agents are first moved to u 's parent. Subsequently, u is removed according to the “safe” manner mentioned before.
- (b) Otherwise, (if the request was not to delete a node) the requested event takes place when the the request is granted a permit.

After the above is performed, the agent terminates.

If there is no static package at u then the following happens.

3. The agent applies the *Up* command repeatedly until reaching either the root or a node $\rho(u)$ that is a filler node with respect to u at that time (as defined in Section 3) or some node x marked ‘locked’, or a node that contains a reject package. To detect a filler node, whenever the agent reaches a new node w , it queries the taxi for the value $d(u, w)$ using the command *Distance*. Consider the following cases.

- (a) The case where x is marked ‘locked’ is described in item 1a above.
- (b) The case that x contains a reject package is described in item 1b above.
- (c) Otherwise (x is not marked ‘locked’ and does not contain a reject package), the agent acts as follows.
- (d) If the node is a filler $\rho(u)$ (with respect to u) then let $P(u)$ be the mobile package of level $j(u)$ residing at $\rho(u)$ as described in the definition of a filler node above.
- (e) If the agent reaches the root and the root is not a filler node (with respect to u) then let $j(u)$ be the smallest integer $j(u) \geq 0$ such that $d(u, r) \leq 2^{j(u)+1}\psi$. In this case, the agent writes at the whiteboard of the root a new package $P(u)$ of size $2^{j(u)}\phi$ and indicates in the package’s flag that $P(u)$ is mobile. Then, the agent increases the value of the variable **Issued** (written at the root’s whiteboard) by $2^{j(u)}\phi$. In addition, if consequently $M < \mathbf{Issued}$ then the agent creates a reject agent, and continues only after the reject agent finished its action on the root. As described later, by that time, the root contains a reject package, and the agent acts as in item 1b.

4. After performing either item 3e or item 3d above, the following happens.

For each $k \in \{0, 1, 2, \dots, j(u) - 1\}$, let u_k be the ancestor of u satisfying $d(u, u_k) = 3 \cdot 2^{k-1}\psi$. The agent applies the following procedure $\text{PROC}(P)$ recursively, starting with the mobile package $P = P(u)$ of level $j(u)$ at its hosting node.

$\text{PROC}(P)$: Given a mobile package P of level k written in the whiteboard of a vertex w , act as follows.

- Erase P from w 's whiteboard and put k inside the variable *Bag* of the agent.
- If $k > 0$ then the agent invokes the instruction *Down* repeatedly until reaching node u_k (note that the agent can recognize u_k by querying the taxi with the instruction *Distance*, and using the value k in its *Bag* variable). Then the agent writes in u_k 's whiteboard two mobile packages P_1 and P_2 , each of level $k - 1$. Also, the agent empties its *Bag* variable, and applies the procedure recursively with P_2 .

- If $k = 0$ (including the case where $j(u) = 0$), then the agent repeatedly invokes the instruction *Down* until reaching node u (note that the agent can recognize u by querying the taxi with the instruction *Distance*).

The request is then granted to u from the level zero package P as described in item 2 above.

The agent then walks up again to the topmost node it ever reached, using the command *Up* and the query command *DistToTop*.

The agent then moves back to u using the instruction *Down* and the query *Distance*. On its way down from x to u , the agent sets the state of every vertex passed (including u) to 'unlocked'.

Reject agents: Recall that in item 3e above, the root may create a reject agent. The reject agent at a node with $s > 0$ children applies the following *reject procedure*: it creates s reject agents, one per child, each carrying (in a *Bag* variable) a reject package (representing infinitely many rejects). Each of these new s reject agents moves to a different child of u , placing the reject package there, and, if the child node has children, the corresponding agent applies the reject procedure recursively.

B.2 Implementing the taxi algorithm

The implementation is rather straightforward, so the details are left to the full version. Let us mention a few hints here. First, when the tree is built, we assume that each node remembers its port number leading to its parent. Therefore, the *Up* command can be easily implemented by the taxi algorithm. Second, when an agent arrives at a node, we assume that the node can detect on which link the agent arrived. This information is accessible to the taxi algorithm in that node. If the agent becomes passive (waiting for the node to become unlocked) then the taxi algorithm in that node u keeps the agents in the agents queue of u .

For an agent created at u , the taxi algorithm carries with the agent also the counter *Distance*, containing the value of the distance from the agent to u . In particular, when the agent is created in u , the taxi algorithm at u sets this value to zero. When the agent arrives at a new node following the execution of an *Up* action, the taxi algorithm in the new node increases this value by one. When the agent arrives at a node following a *Down*(u) operation, the taxi algorithm at the new node deducts one from this value. Counter *Distance* enables the taxi algorithm to answer the *Distance* query from the agent. When the agent is at the root or at the filler node with respect to u , then the taxi algorithm initializes its counter *DistToTop* to zero. The taxi then updates the value of *DistToTop* similarly to the way it maintains *Distance*.

When the agent locks a node, the taxi algorithm saves in the node the pointer to the edge leading to the child from which the locking agent arrived. (For other agents that may reside in the locked node, their arrival ports are not remembered in the whiteboard, but are still carried in the agents). Add to this the fact that all the nodes on the route of the agent from u to the agent's current host node are locked. Hence, the taxi algorithm has enough information to perform the *Down* command (that is, to know to which child to send the agent on the way down).

B.3 Correctness and complexity

We start with two observations. The size of the *Bag* carried by an agent is the size of a package, which is $O(\log M)$. In addition, the taxi counts up to n for each agent, to implement *Distance* and *DistToTop*. The rest of the memory needed per agent is constant. Because of the locking mechanism, each node may contain up to one agent per child. In addition, every node contains $O(M)$ packages for each of the $O(\log n)$ levels (plus some constant per package), this means $O(\log n \log M)$ memory for the packages. In addition, the root contains the variable **Issued**, which is of size $O(\log M)$. This leads to the following observation (assuming $M > n$):

Observation B.1 *The memory required by the distributed controller per node is $O(\log m \log n)$.*

The next observation is needed for the correctness.

Observation B.2 *In any execution of the distributed algorithm, every agent returns to its origin eventually, and every request is answered (either by a permit or by a reject).*

Proof: Recall, that agents lock nodes only on the way towards the root. Hence, no cycle can be created in the chain of agents waiting for each other. Recall also, that agents waiting in a node are served according to the First In First Out discipline. Moreover, since local computation take zero time, the first agent waiting at the node is dequeued before any new agent reaches the node from another node. Therefore, no deadlock and no starvation can arise. I.e., every request is eventually either granted or rejected. ■

Lemma B.3 *Let S be a scenario of inputs for the distributed controller and let $E_D(S)$ be an execution of the distributed algorithm on the scenario S . There exists a second scenario S' and an execution $E'_D(S')$ of the distributed algorithm on the scenario S' , satisfying the following conditions.*

1. *Each request in S' is initiated after all updates in $E'_D(S')$ regarding the previous request have been completed.*
2. *The dynamic data structures resulted by executions $E_D(S)$ and $E'_D(S')$ are the same.*
3. *The number of messages used during the executions $E_D(S)$ and $E'_D(S')$ is asymptotically the same.*

Proof: We say that a request is *irrelevant* if the request arrives at a node u that contains a reject package. Recall, that we assumed that while a previous request of the same node u is still been treated by an agent of u , a new request cannot arrive at u from the environment. If such messages arrive eventually when node u already contains a reject package, then they become irrelevant.

Note, that from the definition of the problem, that scenario S of inputs may consist of an infinite sequence of requests. However, the number of requests in S that are not irrelevant is finite. Let $\mathcal{R}(E_D(S))$ be the set of requests in S that are not irrelevant in $E_D(S)$. Let $t_{max}(E_D(S))$ denote the first time after which no package is moved in execution $E_D(S)$. Note, that all the requests in $\mathcal{R}(E_D)$ arrive before time $t_{max}(E_D)$. Moreover, all the updates made in the execution $E_D(S)$ in response to these requests are completed by time $t_{max}(E_D)$.

For a request $R \in \mathcal{R}(E_D(S))$, let $P(R)$ be the subpath of nodes that were ever locked by the agent of u . Note, that $P(R)$ is a (non-empty) path connecting u , the *Origin* of R with one of u 's ancestors (possibly u itself). Let R_{last} be the request that is the last to receive a reply (either a reject or a permit) in $\mathcal{R}(E_D(S))$. It is immediate from the distributed implementation, that in execution $E_D(S)$, request R_{last} is answered (either by a grant or a reject) before the agent of R_{last} unlocks any node in $P(R_{last})$. Therefore, starting from the time the agent of R_{last} locked $w \in P(R_{last})$ in execution $E_D(S)$, no agent in $\mathcal{R}(E_D(S))$ has been waiting in the agents queue of w . (Otherwise, that agent would have answered its corresponding request after request R_{last} was answered). Similarly, no request agent passed through w after the agent of R_{last} unlocked it.

Let s be the number of requests in $\mathcal{R}(E_D(S))$. For $i = 1, 2, \dots, s$, let R_i be the i 'th request and let t_i be the time that R_i arrived (from the environment). Scenario S , projected on $\mathcal{R}(E_D(S))$, can be described as the sequence of pairs $(R_1, t_1), (R_2, t_2), \dots, (R_s, t_s)$. Let j be the index such that R_{last} is R_j . Let \hat{t} be a time after all the updates made by agents corresponding to requests $R_1, R_2, \dots, R_{j-1}, R_{j+1}, \dots, R_s$ were completed in execution $E_D(S)$.

Consider now the scenario \hat{S} : $(R_1, t_1), (R_2, t_2), \dots, (R_{j-1}, t_{j-1}), (R_{j+1}, t_{j+1}), \dots, (R_s, t_s), (R_j, \hat{t})$. It follows from the above discussion and from Observation B.2 that there exists an execution $\hat{E}_D(\hat{S})$ of the

distributed algorithm on scenario \hat{S} such that the dynamic data structure resulted from execution $E_D(S)$ is the same as the one resulted from execution $\hat{E}_D(\hat{S})$. Moreover, the number of messages used during these executions $E_D(S)$ is the same.

It follows by induction that there exists a scenario S' and an execution $E'_D(S')$ of the distributed algorithm on scenario S' , satisfying the second and third conditions mentioned in the lemma. In addition, by adding to S' the request in S which are not in $\mathcal{R}(E_D)$ (all these requests are irrelevant) in their original order and after all updates in $\mathcal{R}(E_D)$ have occurred, the first condition in the lemma is also satisfied. The lemma follows. ■

Lemma B.4 *Let S' and $E'_D(S')$ be a scenario and an execution such as the those whose existence is claimed in Lemma B.3. Let $E_S(S')$ be the sequential execution on scenario S' .*

1. *The operations performed in $E'_D(S')$ on the data structure are the same as the corresponding sequential operations that are performed in $E_S(S')$.*
2. *The message complexity used by an agent of a request $E'_D(S')$ is also the same as the move complexity of the corresponding operation in $E_S(S')$ up to a constant factor.*

Proof: We prove by induction on the order of arrival of requests to $E_S(S')$. The claim holds for an empty $E_S(S')$ trivially. Assume that the claim holds for a prefix of $E_S(S')$ and consider the next request R in $E_S(S')$. Let u be the node to which R arrives. The lemma follows from the property of S' that no action is taken for any other request until S' is answered, from Observation B.2, from the induction hypothesis that the data structures are the same at this point for the executions $E'_D(S')$ and $E_S(S')$, and from following the code of the two algorithms and observing that the actions taken by the agent in the distributed controller in this case are exactly the actions taken by the non-distributed algorithm. In particular, if the agent corresponding to R decides that its hosting node is a filler node with respect to u , then this indeed is the case, since no other agent has acted from the time this agent arrived. Moreover, this is indeed the closest filler node to u , since otherwise the agent would have found the conditions of item 3 to hold earlier.

For the message complexity, note that the messages are used only to move the agent, and to move packages when a node is deleted. The number of deletions is at most U . Moving packages as a result of one deletion costs $O(\log^2 n)$ (it may be more than 1, since the size of one package is $O(\log n)$ bits; the deleted node may have held up to $O(n)$ packages of each level.) An agent that is locked in the deleted node has to be moved to the parent too. This costs an additional message per edge (since an agent who is waiting at u first locked a child of u).

For agents moves, note, that in the case that the agent reached a filler node or the root, it traversed only three times the distance from u to the filler node. ■

The lemma below follows directly from the above lemma and from Lemmas B.3, B.4, 3.2 and 3.3.

Lemma B.5 *The distributed controller is correct and its message complexity is $O(U \frac{M}{W} \log^2 U)$.*

As in Lemma 3.4, using iteration, the message complexity can be further reduced. When an iteration terminates, $O(n)$ messages suffice to count the number L of unused permits mentioned for Lemma 3.4. Note, that this is not more than the message complexity of the iteration.

Theorem B.6 *The distributed controller is correct and its message complexity is $O(U \log(\frac{M}{W+1}) \log^2 U)$.*

B.4 The general case

We now describe how to handle the general case, where it is not assumed that an upper bound U on the number of nodes ever existing in the graph is given in advance. We give a distributed implementation to the algorithm described in Subsection 3.3 and the proof of (the first part of) Theorem 3.5.

Let us first explain how to implement the actions of that algorithm. The implementation of one iteration is just the distributed implementation described above. Recall, that an iteration is stopped when Z_i - the number of topological changes, reaches a certain value. To implement that in the distributed setting, we use a second controller (for known U , as described above) that counts only the topological changes. Hence, the root has an estimate of Z_i . Indeed, in the proof of Theorem 3.5, we assumed that Z_i is known exactly, but it is not difficult to change the proof slightly so that an estimate is good enough. Hence, the root knows when to end the iteration.

The algorithm also needs to know Y_i - the exact number of permits granted, and N_{i+1} . This is implemented by a distributed wave and echo (a broadcast and convergecast) initiated by the root at the end of the iteration. Finally, the algorithm needs to reset the data structure at all the nodes. This too is done by the wave and echo. Note, that the message complexity of the wave and echo is at most a constant factor times the message complexity used in the iteration. In addition, the fact that we are running two controllers in parallel in each iteration only increases the message complexity by a constant factor. Hence, this distributed implementation of the proof of Theorem 3.5 has the same message complexity (up to a constant factor) as the move complexity stated there.

Regarding the message size, we note that an agent walking up the tree at some time t , needs only to remember the number of steps k it passed from its origin. Since all the nodes on the path connecting the agent and its origin are locked, the current number of nodes in the tree n is at least k , and therefore, the agent can be encoded using $O(\log n)$ bits. Similar argument holds also for the case that the agent goes down the tree, towards its origin. It follows, that all messages can be encoded using $O(\log n)$ bits, where n is the current number of nodes in the tree.

The above discussion sketched the proof for the following theorem, that is the distributed equivalent of Theorem 3.5.

Theorem B.7 *There exists a distributed controller whose move complexity is $O(n_0 \log^2 n_0 \cdot \log \frac{M}{W+1}) + O(\sum_j \log^2 n_j \cdot \log \frac{M}{W+1})$, where n_j is the number of nodes immediately after the j 'th topological change occurs.*

Corollary B.8 *The estimator has message complexity $O(n_0 \log^2 n_0 + \sum_i \log^2 n_i)$, where n_i is the number of nodes immediately after the i 'th topological event takes place.*

C Additional applications of the controller presented in this paper

We demonstrate the use of the new controller for extending various existing distributed data structures for *local queries*. Constructing the data structure, or maintaining it when the network graph changes, requires communication. However, given the data structure, when a node u asks a query, it receives an answer without using any communication. In the query, u may specify some other node v . Examples are routing (“which neighbor of u is the next on the route to v ?”), distance (from u to v), etc. The efficiency of such a scheme is measured in terms of the size of the memory required in each node and in terms of the number (and sizes) of messages required to update the data structure if the network graph changes. Multiple such data structures are described in the literature, for static networks, or for limited dynamic networks. Using the new controller it is possible to extend many such data structures to operate efficiently also under controlled deletions of leaves and sometimes also of internal nodes.

Consider, for example, a case where deletions of a certain type do not affect the correctness of the data structure. For example, deletions of degree one vertices do not affect the distance between existing

nodes, therefore, the correctness of a given static distance labeling scheme (such as the ones in [17]), is not affected by such deletions. At first glance, it may seem that extending such a data structure to support also deletions of that type is trivial. Given such a data structure D that does not support deletions, just use it when deletions are allowed. I.e., in the case of an allowed event that is not a deletion (say, an insertion of a leaf), take the same update action taken in D , and in the case of a deletion event, perform no data structure update action (though the deletion of the node itself is performed). Unfortunately, this approach is no longer efficient in terms of the size of the memory required in a node. For example, consider a large graph with optimal size routing tables T^L . If the number of nodes decreases significantly because of deletions, the optimal size routing tables T^S for the resulting smaller graph are much smaller than T^L . If the algorithm that maintains the routing tables (in this specific example) does not take any action for deletions, then the data structure stays with tables of size T^L instead of the new optimal T^S . Using the new controller, it is possible to estimate the number of nodes, and recompute the data structure when the graph shrinks significantly. (Of course, recomputation can be performed after every deletion, but that would be inefficient in terms of the number of messages).

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

Claim C.1 *The correctness of the following types of data structures are not affected by deletions of either internal nodes or leaves. (For a dynamic routing scheme, if the destination vertex u of a message is deleted, then no requirements are imposed on that message, in particular, it is not required that the message reaches u .)*

- *Any exact (stretch 1) routing scheme (either dynamic or static and either labeled or name-independent) on trees. (E.g., the dynamic routing schemes in [15, 19, 26] or the static schemes in [16, 28])*
- *Any ancestry labeling scheme (either dynamic or static) on trees. (E.g., the dynamic ancestry schemes in [19], and the static ones in [18]).*

Claim C.2 *The correctness of the following types of data structures are not affected by deletions of degree one vertices.*

- *Any exact (stretch 1) routing scheme (either dynamic or static and either labeled or name-independent) on general graphs.*
- *Any labeling scheme on any type of graph family (closed under deletions of leaves) which supports either the distance or the flow or the k -vertex connectivity functions. (E.g., the distance labeling schemes in [17] and the flow and vertex connectivity labeling schemes in [20, 22, 21]).*
- *Any nearest common ancestor (NCA) labeling scheme (either dynamic or static) on trees.*

For a dynamic labeling schemes, some studies distinguish between two kinds of memory. One is used for the label $\mathcal{L}(v)$ given to each node v to deduce the required information in response to online queries. The other is used during updates and maintenance operations. See, e.g. [26, 19, 25]. For certain applications (and particularly routing), the label $\mathcal{L}(v)$ is often kept in the router itself and used frequently, providing fast calculations of the routes. On the other hand, the additional storage $Memory(v)$ may be kept on some external storage device, and possibly used less frequently and less urgently. This means that the size of $\mathcal{L}(v)$ seems to be a more critical consideration than the total amount of storage needed for the information maintenance.

Lemma C.3 *Let π be one of the (either static or dynamic) data structures mentioned in either the above claim, and let $f(n)$ be an upper bound on the size of the labels (or routing tables) $\mathcal{L}(v)$ used in π . Assume that $f(n)$ is a reasonable² function. Let $\mathcal{M}(\pi, n)$ be the maximum message complexity used to assign the labels (or routing tables) of π on an n -node network. Then the following holds.*

²A reasonable function is a function $f(n)$ satisfying that there exists a constant c , such that for any $n/2 < m < n$, $f(n) \leq c \cdot f(m)$. This condition is satisfied by a function of the form $f(n) = \alpha n^\epsilon \log^\beta n \log^\gamma \log n$, for $\alpha, \epsilon, \beta, \gamma > 0$

1. If π be one of the schemes mentioned in Claim C.1, then there exists an extended scheme supporting also controlled deletions of both leaves and internal nodes. The extended scheme has label size $O(f(n))$ and addition additive factor of $O(\max\{\frac{\mathcal{M}(\pi, n)}{n} \mid n > 0\})$ to the amortized message complexity of π , per topological change.
2. If π be one of the schemes mentioned in Claim C.2, then there exists an extended scheme supporting also controlled deletions of degree one vertices. The extended scheme has label size $O(f(n))$ and addition additive factor of $O(\max\{\frac{\mathcal{M}(\pi, n)}{n} \mid n > 0\})$ to the amortized message complexity of π , per topological change.

Corollary C.4 *The routing and ancestry schemes on trees mentioned in [15, 26, 16] can be extended to support also controlled deletions of both leaves and internal nodes. The extended scheme has the same asymptotic label size of the original scheme, and its message complexity is $O(n_0 \log^2 n_0) + O(\sum_j \log^2 n_j)$, where n_j is the number of nodes in the tree immediately after the j 'th topological change.*