New Bounds for the Controller Problem

Yuval Emek * Amos Korman †

Abstract

The (M, W)-*controller*, originally studied by Afek, Awerbuch, Plotkin, and Saks, is a basic distributed tool that provides an abstraction for managing the consumption of a global resource in a distributed dynamic network. The input to the controller arrives online in the form of *requests* presented at arbitrary nodes. A request presented at node u corresponds to the "desire" of some entity to consume one unit of the global resource at u and the controller should handle this request within finite time by either *granting* it with a *permit* or *denying* it. Initially, M permits (corresponding to M units of the global resource) are stored at a designated *root* node. Throughout the execution permits can be transported from place to place along the network's links so that they can be granted to requests presented at various nodes; when a permit is granted to some request, it is eliminated from the network. The fundamental rule of an (M, W)-controller is that a request should not be denied unless it is certain that at least $M - W$ permits are eventually granted. The most efficient (M, W)-controller known to date has message complexity $O(N \log^2 N \log \frac{M}{W+1})$, where N is the number of nodes that ever existed in the network (the dynamic network may undergo node insertions and deletions).

In this paper we establish two new lower bounds on the message complexity of the controller problem. We first prove a simple lower bound stating that any (M, W)-controller must send $\Omega(N \log \frac{M}{W+1})$ messages. Second, for the important case when W is proportional to M (this is the common case in most applications), we use a surprising reduction from the (centralized) *monotonic labeling problem* to show that any (M, W)-controller must send $\Omega(N \log N)$ messages. In fact, under a long lasting conjecture regarding the complexity of the monotonic labeling problem, this lower bound is improved to a tight $\Omega(N \log^2 N)$. The proof of this lower bound requires that $N = O(M)$ which turns out to be somewhat inevitable due to a new construction of an $(M, M/2)$-controller with message complexity $O(N \log^2 M)$.

*School of Electrical Engineering, Tel Aviv University, Tel Aviv, 69978 Israel. E-mail: yuvale@eng.tau.ac.il. Supported in part by the Israel Science Foundation, grant 664/05.

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

1 Introduction

1.1 Background

A centralized online algorithm typically makes decisions based on past information, lacking any knowledge of what the future holds. In a distributed setting the input is spread over distant nodes in a network, hence introducing an additional kind of uncertainty, where nodes should make decisions based on local information without knowing what already happened in remote parts of the network. This paper addresses a basic problem which is affected by both kinds of uncertainties: controlling the consumption of a global resource. (For other problems that deal with both kinds of uncertainties, see, e.g., [5, 20].)

Consider for example the case in which some finite amount of money (the global resource) resides somewhere in the network (in one node, or in several), and occasionally different nodes wish to withdraw a certain amount of money. A withdrawal request made by node u is either *granted*, in which case the requested amount of money is transferred to u (a portion of the global resource is consumed), or *rejected*. We are interested in a *distributed bank* protocol that handles these withdrawal requests while guaranteeing that a request is not rejected if there is still enough money available in the network.

Controllers (originally studied in [1] and later in [17]) provide an abstraction for such a distributed bank protocol and more generally, for global resource consumption management. Considered as one of the elementary and fundamental tools in distributed computing (cf. [2]), controllers serve as a key ingredient in the state of the art solutions for various problems such as majority commitment in a network where some of the nodes failed before the algorithm started [3, 6, 13, 21], routing (and other informative labeling problems) in dynamic trees [15, 16, 18, 19], and dynamic name assignment [1, 16, 17].

The (M, W)-*controller* problem. We consider a distributed network operating in an asynchronous environment. Initially, a set of *permits* resides at some designated node called the *root*. A subset of permits may be delivered from node u to any of its neighbors v by sending a single message from u to v (this message essentially encodes the number of permits that are being delivered). Therefore throughout the execution the permits are distributed among the nodes of the network and different nodes may hold different numbers of permits. The input to the controller arrives online in the form of *requests* presented at arbitrary nodes. When a request is presented at node u, the controller must respond within finite time in one of the following two manners: (1) it may *grant* the request by delivering a permit to u in which case the permit is eliminated from the network (corresponding to consuming one unit of the global resource at node u); or (2) it may *reject* the request.

In an (M, W)-controller, the number of permits that initially reside at the root is M, indicating

that at most M requests can be granted. On the other hand, the (M, W)-controller may reject a request only if it is certain that at most W permits eventually remain in the network. In other words, if an (M, W)-controller rejects a request, then it is guaranteed that at least $M - W$ requests were already granted (or will be granted within finite time).

It is assumed in [1, 17] that a spanning tree T rooted at some node r is maintained in the network and that the controller relies on the links of T for communication. The global resource whose consumption is managed by the controller may be of various types. However, since the concept of an (M, W)-controller finds many applications in dynamic networks, a special attention has been given to the case where a request presented at node u represents the desire to perform a topology change at the vicinity of u. Such a request is referred to as a *topological request*. Specifically, the topology changes considered in this context are: (i) inserting a new child of u as a leaf in T; (ii) inserting a new child of u as an internal node in T by subdividing a link that connects u to one of its children; and (iii) deleting a child v of u and turning the children of v into children of u (the root r is never deleted). In all three cases the actual topology change is assumed to occur once the topological request is granted a permit[1].

The number of nodes that ever existed in the network (including the deleted ones) is denoted by N. Note that N cannot exceed the initial network size by more than M since the insertion of every new node should be granted a permit by the controller (in fact, the combined number of node insertions and deletions is at most M).

The efficiency of an (M, W)-controller is measured by means of its *message complexity*, namely, the total number of messages sent during the execution. This is usually expressed as a function of M, W, and N. Consider for example the following naive implementation for an (M, W)-controller. Upon receiving a request at node u, the naive controller sends a message to the root r asking for a permit. The root returns a permit in response to each of the first $M - W$ arriving messages; afterwards, it broadcasts some "out of permits" message to all nodes, so that subsequent requests are rejected with no further consideration. Exchanging messages between u and r in an N-node network may require $\Omega(N)$ messages, hence the message complexity of this naive (M, W)-controller can be as large as $\Omega(N(M - W))$ even if the requests are spaced in time so that each request is granted before the next request is presented (which is typically far from being the case in an asynchronous network).

The monotonic labeling problem. Vital to our techniques is the *monotonic labeling problem*. In this (centralized) problem n distinct elements from some dense totally ordered set S (e.g., the real numbers) are introduced, one at a time. Upon introduction, each element $x \in S$ should be

[1] The protocols responsible for executing the actual topology change may be interesting by their own right, however, for simplicity, previous works ignored the details of these protocols by assuming that the requesting entity is taking care of performing the topology change. For further details regarding the dynamic model and its applications see [17].

assigned with a *label* $\lambda(x)$ taken from some discrete totally ordered set L of adequate ($|L| \geq n$), yet limited, cardinality (e.g., the integers $1, \ldots, |L|$). The order of the labels must agree with the order of the elements, that is, for every two elements $x, y \in S$, if $x < y$, then $\lambda(x) < \lambda(y)$. Therefore from time to time some previously introduced elements must be *relabeled* to "make room" for new elements. The objective function of an algorithm for the monotonic labeling problem is to minimize the total number of labeling operations (including relabeling previously introduced elements). This is typically measured as a function of n and with respect to the cardinality of the label set L (clearly, the problem becomes easier as $|L|$ grows).

1.2 Related work

The most relevant works to this paper, are the works of [1] and [17]. In [1], Afek et al. construct the first (M, W)-controller which admits message complexity $O(N \log^2 N \log \frac{M}{W+1})$. It is based on the following principle. The M permits, which initially reside at the root, are disseminated and moved by the controller in order to grant arriving requests. At any time, the remaining permits are stored at specific bins which are organized according to an underlying structure called the *bin hierarchy*. This bin hierarchy is employed in order to preserve some "sparseness" properties of the distribution of the remaining permits which are essential for the analysis. In terms of topology changes, the controller of Afek et al. only supports the insertion of leaves.

Korman and Kutten [17] introduce an (M, W)-controller with a similar message complexity[2] which supports all three types of topology changes (i.e., the insertion of leaves, the insertion of internal nodes, and the deletion of nodes). The improvement is achieved by relaxing the hierarchy of bins and constructing it on the fly, in a more local fashion.

Both the (M, W)-controller of [1] and that of [17] are implemented by first constructing an $(M, M/2)$-controller with message complexity $O(N \log^2 N)$, and then invoking it in $O(\log \frac{M}{W+1})$ iterations. Observe that the iterative argument does not hold if W is large so that $\log \frac{M}{W+1} = o(1)$. Indeed, it is implicitly assumed in [1, 17] that $W \leq M(1 - \Omega(1))$. The controller of [17] encodes each message using $O(\log N)$ bits, while the (more restricted) controller of [1] encodes each message using $O(\log \log N)$ bits.

On the negative side, it is easy to see that an $\Omega(N)$ term in the message complexity of any (M, W)-controller is inevitable. (In the case of an N-node path, for example, merely delivering a permit from the root to a request presented at the other end requires N messages.) However, no non-trivial lower bounds were previously known.

[2] The message complexity of the protocol of [17] is actually sometimes slightly better than $O(N \log^2 N \log \frac{M}{W+1})$. The total number of messages sent by that protocol is $O(N_0 \log^2 N_0 \log \frac{M}{W+1}) + O(\sum_i \log^2 N_i \log \frac{M}{W+1})$, where N_0 is the initial number of nodes in the network and N_i denotes the number of nodes after the i^{th} topology change occurs. The parameter N can be thought of as N_0 plus the number of node insertions. Note, that if $M < N_0$ then the message complexity of both the controller of [17] and the controller of [1] is $O(N \log^2 N \log \frac{M}{W+1})$.

The monotonic labeling problem is essentially introduced in [14] and studied further in [9, 23, 22, 11, 4, 12, 7, 10], mainly in the context of maintaining an ordered data structure. With label sets of cardinality n, $n(1+\epsilon)$, and $n^{1+\epsilon}$, where ϵ is any positive constant, the known upper bounds for the number of labeling operations are $O(n \log^3 n)$ [4], $O(n \log^2 n)$ [14, 23, 7], and $O(n \log n)$ [9, 22, 11]. An $\Omega(n \log n)$ lower bound for the number of labeling operations with label sets of cardinality polynomial in n is established in [10], thus showing that the upper bound of [9, 22, 11] is tight. Based on a lower bound established in [12] for the special class of *smooth* algorithms, the authors of [12, 10] conjecture that any monotonic labeling algorithm with $O(n)$ labels requires $\Omega(n \log^2 n)$ labeling operations, hence the upper bound of [14, 23, 7] is also tight.

1.3 Our contribution

In this paper we establish new bounds on the message complexity of the controller problem. As a warm up, we first prove a simple lower bound stating that any (M, W)-controller must send $\Omega(N \log \frac{M}{W+1})$ messages. Although this lower bound is meaningful for small values of W, it is not very informative when W is proportional to M, which is the typical case in many applications of the controller problem.[3]

Subsequently, we turn our attention to the case where W is proportional to M and prove that for every constant $\epsilon > 0$, an $(M, M(1-\epsilon))$-controller on a dynamically growing path of initial size M must admit message complexity $\Omega(M \log M) = \Omega(N \log N)$. This lower bound is obtained due to a surprising reduction from the (centralized) monotonic labeling problem to the (distributed) controller problem. Through this reduction, the $\Omega(n \log n)$ lower bound on the number of labeling operations that must be performed by any monotonic labeling algorithm with a label set of cardinality polynomial in n translates to the desired $\Omega(N \log N)$ lower bound on the message complexity of a controller. In fact, the reduction holds for monotonic labeling algorithms with label sets of cardinality $O(n)$, and therefore as it turns out, under the conjecture of [12, 10], we obtain a tight $\Omega(N \log^2 N)$ lower bound on the message complexity of any $(M, M(1-\epsilon))$-controller.

Both our lower bounds hold even when the message size is unbounded. Furthermore, they do not rely on concurrency considerations, and therefore remain valid even if the system is synchronous and the requests are "spaced in time" so that the next request is presented only after the controller finished handling all previous ones.

As previously mentioned, the proof of the $\Omega(N \log N)$ lower bound (and also of the conjectured tight $\Omega(N \log^2 N)$ lower bound) relies on a network of initial size M which, in particular, implies that $N = \Theta(M)$. It turns out that this is no coincidence: such a lower bound cannot hold if M is much smaller than N. We prove it by constructing a novel $(M, M/2)$-controller with message

[3] In particular, the case $W = M/2$ is the one used to derive the state of the art solutions for the routing problem (and other labeling problems) on dynamic trees [15, 16, 18, 19] as well as for the dynamic name assignment problem [1, 17].

complexity $O(N \log^2 M)$. Apart from demonstrating the inherent limitation of our lower bound proof technique, the new controller is interesting as it can be generalized (c.f. Section 5 in [1]) to an (M, W)-controller with message complexity $O(N \log^2 M \log \frac{M}{W+1})$, thus exhibiting an asymptotic improvement to the state of the art in the case that M is sub-polynomial in N. Moreover, the structure of our new controller is completely different than the previously known controllers and bears an independent algorithmic interest.

2 Lower bounds

2.1 The $\Omega(N \log \frac{M}{W+1})$ lower bound

We begin the technical part with a simple lower bound that provides a good demonstration of the definition of a controller. Let P be an N-node (static) path and let π be any (M, W)-controller that supports non-topological requests on P. We prove that there exists a scenario Γ that forces π to send $\Omega(N \log \frac{M}{W+1})$ messages. Note that if $\log \frac{M}{W+1} = O(1)$, then the required lower bound is dominated by the trivial $\Omega(N)$ lower bound. We may therefore assume that $\log \frac{M}{W+1} = \omega(1)$. Moreover, we assume for simplicity that both $M + 1$ and $W + 1$ are powers of 2. (The proof can be easily modified to handle an arbitrary choice of parameters.)

Let u and v be the two end nodes of P. The desired request sequence Γ admits the following two features. First, each request in Γ is presented after all actions of π in response to the previous request are completed. Second, each request is presented at either u or v. The sequence Γ is divided to $\lambda = \log \frac{M+1}{W+1} - 1$ subsequences denoted $\Gamma = \gamma_1 \cdot \gamma_2 \cdots \gamma_\lambda$. For every $1 \leq i \leq \lambda$, the i^{th} subsequence γ_i consists of $(M + 1)/2^i$ requests which are presented (all of them) either at u or at v. The proof relies on designing the request subsequences γ_i so that in response to each one of them, π must send $\Omega(N)$ messages.

We construct the request subsequences γ_i, by induction on i. Let γ_0 denote the empty subsequence. Given $1 \leq i \leq \lambda$, assume that the prefix $\Gamma_{i-1} = \gamma_0 \cdot \gamma_1 \cdots \gamma_{i-1}$ is already determined and construct the subsequence γ_i as follows.

Let $\gamma(u)$ (respectively, $\gamma(v)$) denote a sequence of $(M + 1)/2^i$ requests presented at u (resp., at v). Consider the subsequences

$$\Gamma(u) = \Gamma_{i-1} \cdot \gamma(u) \quad \text{and} \quad \Gamma(v) = \Gamma_{i-1} \cdot \gamma(v) \ .$$

Observe, that both $\Gamma(u)$ and $\Gamma(v)$ contain

$$(M + 1)(1 - 1/2^{i-1}) + (M + 1)/2^i = (M + 1)(1 - 1/2^i) < M - W$$

requests (the last inequality follows from the fact that $i < \log \frac{M+1}{W+1}$). Therefore π cannot deny any request in response to either $\Gamma(u)$ or $\Gamma(v)$.

Now, consider the request sequence

$$\Gamma(u,v) = \Gamma_{i-1} \cdot \gamma(u) \cdot \gamma(v) \ .$$

As $\Gamma(u,v)$ contains

$$(M+1)(1-1/2^{i-1}) + 2(M+1)/2^i = M+1$$

requests, at least one of them should be denied by π. This means that π somehow "distinguishes" $\Gamma(u,v)$ from both $\Gamma(u)$ and $\Gamma(v)$. More formally, after handling the prefix Γ_{i-1}, either $\Omega(N)$ messages are sent in response to $\gamma(u)$ or $\Omega(N)$ messages are sent in response to $\gamma(v)$ (or both). If the former is true, then we fix $\gamma_i = \gamma(u)$; otherwise, we fix $\gamma_i = \gamma(v)$.

To summarize, our construction of the request sequence $\Gamma = \gamma_1 \cdots \gamma_\lambda$ guarantees that π sends $\Omega(N)$ messages for each $1 \leq i \leq \lambda$. This sums up to $\Omega(N \log \frac{M}{W+1})$ in total.

2.2 The $\Omega(N \log N)$ lower bound

We now turn to prove the main result of the paper, namely, that for every constant $\epsilon > 0$, an $(M, M(1-\epsilon))$-controller on a dynamically growing path of initial size M must send $\Omega(M \log M) = \Omega(N \log N)$ messages (recall that N is proportional to M when the initial size of the network is M).

Our method is based on reducing the monotonic labeling problem to the controller problem. Specifically, we show that an $(n, n(1-\epsilon))$-controller that supports node insertion requests on a path of initial size n with message complexity $f(n)$ implies an algorithm for the monotonic labeling of n elements with label set of cardinality $2n$ that performs $O(f(n))$ labeling operations. It is known that such a monotonic labeling algorithm does not exist unless $f(n) = \Omega(n \log n)$ [10] and it is conjectured that $f(n)$ must be $\Omega(n \log^2 n)$ [12, 10]. This implies the following theorem.

Theorem 2.1. *The existence of an $(n, n(1-\epsilon))$-controller with message complexity $f(n)$ for a path of initial size n implies $f(n) = \Omega(n \log n)$ ($f(n) = \Omega(n \log^2 n)$ under the conjecture of [12, 10]).*

To prove Theorem 2.1, consider some instance of the monotonic labeling problem on n elements with label set $\{1, \ldots, 2n\}$. Let $x_1, \ldots, x_n$ denote the n elements in order of introduction. We label the first $\lceil 1/\epsilon \rceil$ elements $(x_1, \ldots, x_{\lceil 1/\epsilon \rceil})$ arbitrarily (since ϵ is constant, this incurs $O(1)$ labeling operations) and deal with the remaining elements in iterations. Let n_i denote the number of elements which were already introduced (and labeled) at the beginning of iteration i ($n_1 = \lceil 1/\epsilon \rceil$), so that the elements introduced during this iteration are $x_{n_i+1}, \ldots, x_{n_{i+1}}$. We label these $n_{i+1} - n_i$ elements in accordance with the execution of an $(n_i, n_i(1-\epsilon))$-controller invoked on a path P. This is done as follows.

At all times, the size of P corresponds to the number of elements that were already introduced. In particular, at the beginning of iteration i we have $|P| = n_i$. Consider the path $P = (u_1, \ldots, u_k)$ after the elements $x_1, \ldots, x_k$ were introduced for some $n_i \leq k \leq n_{i+1}$. The nodes of P are mapped

from left to right to the elements $x_1, \ldots, x_k$ according to their rank, that is, u_j is mapped to the j^{th} smallest element in $x_1, \ldots, x_k$. Let $x(u_j)$ denote the element in $x_1, \ldots, x_k$ to which node u_j is mapped. Note that $x(u_j) < x(u_{j+1})$ for every $1 \leq j < k$.

The labels $\lambda(\cdot)$ assigned to $x_1, \ldots, x_k$ are determined by the permit distribution along the path $P = (u_1, \ldots, u_k)$. For every $1 \leq j \leq k$, the element $x(u_j)$ is assigned with the label $\lambda(x(u_j)) = j + p_j$, where p_j denotes the number of permits stored in the subpath $(u_1, \ldots, u_{j-1})$. Note that this is a valid labeling scheme since it guarantees that
(i) $\lambda(x(u_j)) < \lambda(x(u_{j+1}))$; and
(ii) all labels are taken from the set $\{1, \ldots, 2n_i\} \subseteq \{1, \ldots, 2n\}$.
(To verify that (ii) holds, observe that the sum of $|P|$ and the number of permits stored in P is $2n_i$ throughout the iteration.)

Consider the path $P = (u_1, \ldots, u_k)$ at some stage of iteration i ($n_i \leq k \leq n_{i+1}$). Upon introduction of the next element x_{k+1}, we present a node insertion request to P in a position that corresponds to the rank of x_{k+1} in $x_1, \ldots, x_{k+1}$. If this request is granted, then a new node is inserted into P and x_{k+1} is labeled in accordance with the above scheme (which may cause some relabeling of previously introduced elements). Otherwise (the request is rejected), iteration i is halted, $n_{i+1} \leftarrow k$, and iteration $i+1$ starts by invoking an $(n_{i+1}, n_{i+1}(1-\epsilon))$-controller on a path of initial size n_{i+1}, where the first request corresponds to the insertion of a node mapped to x_{k+1} (recall that this was rejected in iteration i).

Let l be the index of the last iteration (in which element x_n was labeled). For every $1 \leq i < l$, we know that the $(n_i, n_i(1-\epsilon))$-controller that operates in iteration i does not reject any request before at least ϵn_i requests were granted (and that many new nodes were inserted into P), thus $n_{i+1} \geq n_i(1+\epsilon)$. Since $n_1 = \lceil 1/\epsilon \rceil$ and $n_l < n$, we conclude that $l = O(\log n)$.

The algorithm described above is a valid monotonic labeling algorithm: each element is labeled upon introduction and the order of the labels always agrees with the order of the elements. It remains to bound the number of labeling operations performed by our monotonic labeling algorithm. In attempt to do so, we distinguish between two types of labeling operations: (1) those that occur during the execution of one of the controllers; and (2) those that occur when one iteration halts and a new iteration begins (recall that when a new iteration begins the labels of the elements that were already introduced may change). At most n_i labeling operations occur when iteration i begins, hence the total number of labeling operations of type (2) is bounded from above by $\sum_{i=1}^{l} n_i = O(n)$.

We now turn to analyze the number of labeling operations of type (1). Suppose that for every $1 \leq i \leq l$, we have an $(n_i, n_i(1-\epsilon))$-controller that supports node insertion requests on a path of initial size n_i with average message complexity at most $f(n_i)$, where $f : \mathbb{Z}_{>0} \to \mathbb{Z}_{>0}$ is a non-decreasing function. This means that at most $2n_i f(n_i)$ messages were sent in iteration i, which sums up to at most $\sum_{i=1}^{l} 2n_i f(n_i) = O(nf(n))$ messages all together.

The key argument in our analysis is that each type (1) labeling operation accounts for at least

one message sent by the controllers, and hence the total number of type (1) labeling operations is $O(nf(n))$. To justify this argument, consider the path $P = (u_1, \ldots, u_k)$ at some stage of iteration i and observe that the element $x(u_j)$ is assigned with a new label only when the sum S of the number of nodes to the left of u_j and the number of permits stored in these nodes changes. Note that the number of nodes to the left of u_j may increase due to the insertion of a new node in the subpath (u_1, u_{j-1}), but this comes together with the elimination of one permit stored at one of the nodes $\{u_1, \ldots, u_{j-1}\}$. Therefore the sum S changes only when some permits were shifted from $\{u_1, \ldots, u_{j-1}\}$ to $\{u_j, \ldots, u_k\}$ or vice versa which requires the exchange of a message along the path link (u_{j-1}, u_j). It follows that our monotonic labeling algorithm performs $O(nf(n))$ labeling operations in total, thus establishing Theorem 2.1.

3 An (M, W)-controller

In this section we consider a dynamic rooted tree T of initial size N_0 and construct an $(M, M/2)$-controller for T with message complexity $O(N_0 \log^2 M)$ assuming that $M < N_0$. This is done in two stages. First, we reduce the $(M, M/2)$-controller problem from arbitrary trees to simple paths by a novel technique[4] presented in Section 3.1. Therefore the remaining challenge is to construct an $(M, M/2)$-controller with message complexity $O(N_0 \log^2 M)$ for simple paths of initial size $N_0 > M$; this is done in Section 3.2.

3.1 A reduction from trees to paths

In this section we design a transformation from the $(M, M/2)$-controller problem on a tree T to the $(M, M/2)$-controller problem on a path P. A scenario of requests on T is translated under this transformation to an induced scenario of requests on P. An $(M, M/2)$-controller protocol on P handles this (path) scenario and its actions are simulated by the nodes of T. A natural attempt to do so is to map every node in T to a unique node in P (a bijection) so that each tree node simulates the actions taken by its corresponding path node. The efficiency of this method depends on the stretch induced by the mapping: delivering a message from node u to an adjacent node v in P is simulated by delivering a message from u' to v' in T, where u' and v' are the preimages of u and v, respectively. Therefore if u' and v' are far apart in T, then many messages should be sent in T in order to simulate a single message in P and the reduction fails.

Although it is always possible to design a (bijective) mapping that guarantees a constant stretch for every pair of adjacent path nodes, the methods known to us that do so are not very simple to describe (we are unaware of previous works that studied this issue). More importantly, we do not know how to adapt these methods to the dynamic distributed setting. Instead of relying on such

[4] A similar technique was used in [16], where a preliminary version of the current paper is credited.

a bijection, we shall map every node in T to *several* nodes in P. For simplicity of presentation, we shall first describe the desired transformation assuming that all requests are non-topological. In this case, the topology of the tree remains fixed throughout the execution and in particular, the number of nodes N remains unchanged.

At any given time during the execution, the N-vertex rooted tree T is associated with a $2N$-node path P rooted at its leftmost node. This is done by associating each tree node with a pair of path nodes, as follows. Recall that a DFS tour (starting at the root) associates every node u in T with two *timestamps* $1 \leq d[u] < f[u] \leq 2N$, where $d[u]$ records the time step when u was first visited and $f[u]$ records the time step when the examination of u was over (see [8]). (In particular, the root r satisfies $d[r] = 1$ and $f[r] = 2N$ and if u is a leaf, then $d[u] = f[u] - 1$.) Enumerate the $2N$ nodes in P from left to right by the integers $1, \ldots, 2N$ and let each vertex u in T be associated with the nodes $d[u]$ and $f[u]$. Note that each path node x is associated with a single tree node $\mathrm{pre}(x)$, referred to as the *preimage* of x.

A scenario of requests in T is translated to a scenario of requests in P as follows: a request presented at some tree node v is translated to a request presented at the path node $d[v]$. This defines an induced request scenario on P for every request scenario on T. An $(M, M/2)$-controller for P is invoked on this induced request scenario. The actions taken by the path nodes are simulated by their preimages in T. Specifically, if some path node x wishes to send a message to one of its neighbors y in P, then this message is sent from $\mathrm{pre}(x)$ to $\mathrm{pre}(y)$ in T. The permits of the tree controller are subjected to the path controller so that if x delivers some subset of permits to y, then $\mathrm{pre}(x)$ delivers that subset of permits to $\mathrm{pre}(y)$. (Initially, the M permits are stored in the root of the path whose preimage is the root of the tree.) Recall that a request was presented at the path node $d[v]$ under the induced scenario only when a request was presented at the tree node v under the original scenario. If this request is granted a permit (respectively, rejected) by the path controller, then the corresponding request is granted a permit (resp., rejected) by the tree controller. The above simulation clearly implements an $(M, M/2)$-controller on T.

We argue that the distance in T between the preimages $\mathrm{pre}(x)$ and $\mathrm{pre}(y)$ of any two path neighbors x and y is at most 2. Indeed, as demonstrated by Table 1, $\mathrm{pre}(x)$ and $\mathrm{pre}(y)$ are either siblings in T or a child and a parent. Note that in order to simulate the actions of the path controller a tree node u does not have to know the exact DFS timestamps of its associated path nodes $d[u]$ and $f[u]$, but rather the structure of its local neighborhood in T, that is, whether it admits any children, whether it admits a left sibling, and whether it admits a right sibling. The bound on the message complexity of the resulting $(M, M/2)$-controller on the tree T follows.

The case of topological requests follows from a similar approach, but requires some additional technicalities. In Appendix A we provide an overview of this case. A detailed description is deferred to the full version. This completes the proof of the following theorem.

Theorem 3.1. *An $(M, M/2)$-controller on a tree can be implemented by an $(M, M/2)$-controller*

path node	left neighbor	right neighbor
$d[u]$	$f[v]$, where v is the left sibling of u, if u has a left sibling in T; $d[v]$, where v is the parent of u, if u does not have a left sibling in T;	$d[v]$, where v is the leftmost child of u, if u has a child in T; $f[u]$ if u does not have a child in T;
$f[u]$	$f[v]$, where v is the rightmost child of u, if u has a child in T; $d[u]$ if u does not have a child in T;	$d[v]$, where v is the right sibling of u, if u has a right sibling in T; $f[v]$, where v is the parent of u, if u does not have a right sibling in T;

Table 1: The DFS timestamps of the left and right neighbors of some path node.

on a path with asymptotically the same message complexity.

3.2 An $(M, M/2)$-controller for a path

In this section we design an $(M, M/2)$-controller that operates on a path P of initial size N_0. We follow the convention that P is rooted at its leftmost node so that the sole child of a path node is its right neighbor. The controller supports topological requests which means that the path may undergo node insertions and deletions, but since we assume that $M < N_0$, the size of P is $O(N_0)$ at all times. Our controller operates in an asynchronous environment under the *FIFO channel* assumption, that is, if node u sends message m_1 at time t_1 and message m_2 at time $t_2 > t_1$, both to the same neighbor v, then m_1 is received at v before m_2 (this assumption can be easily lifted by using standard acknowledging techniques).

Overview. A central component of our controller is an implicit complete binary tree $\mathcal{T}$ which is simulated by the nodes of P (see also [7]). The height h of $\mathcal{T}$ is proportional to $\log M$. Each vertex x in $\mathcal{T}$ is associated with a subpath P_x of P so that every level of $\mathcal{T}$ induces a pairwise disjoint partition of P. Moreover, if x is a child of y in $\mathcal{T}$, then P_x is a subpath of P_y. The behavior of each vertex x in $\mathcal{T}$ is simulated by some node in P_x.

In a preprocess stage the controller spreads the M permits evenly among the leaves of $\mathcal{T}$. Subsequently, a path node u handles a newcoming request by sending a message to the leaf ℓ such that $u \in P_\ell$ and asking for a permit. While waiting for the permit, u is *locked* which means that it does not handle subsequent requests. A request presented at the locked node u is stored in a queue denoted by $\mathcal{Q}(u)$; when u gets *unlocked*, the request stored at the head of $\mathcal{Q}(u)$ is dequeued and its handling procedure starts. The protocol guarantees that ℓ responds to u's message (by either granting the request with a permit or denying it) within finite time, thus a request cannot remain in the queue indefinitely.

A tree vertex x (that may be a leaf) learns that some nodes in P_x are waiting for permits via an invocation of Procedure `Update` at x. Procedure `Update` first makes sure that all nodes in P_x are locked. Afterwards it counts how many permits remained in the leaves of the subtree $\mathcal{T}_x$ and how many requests in P_x wait for a permit. If the difference is larger than some threshold ρ_i that depends on the level $0 \leq i \leq h$ of x in $\mathcal{T}$, then Procedure `Update` grants permits to the awaiting requests, spreads (evenly) the remaining permits among the leaves of $\mathcal{T}_x$, and unlocks all nodes in P_x. Otherwise, x invokes Procedure `Update` at its parent in $\mathcal{T}$. The threshold ρ_i is designed so that if the root z finds out that the difference of the number of permits in the leaves of $\mathcal{T}_z = \mathcal{T}$ to the number of permit-awaiting requests in $P_z = P$ is smaller than its threshold ρ_h, then more than $M/2$ permits must have been granted and all subsequent requests are rejected.

It is crucial for the analysis of the message complexity that every leaf in $\mathcal{T}$ is assigned with $O(N_0/2^h)$ path nodes. For this purpose, Procedure `Update` is slightly more complicated than what we described in the previous paragraph. Since some of the requests that currently wait for permits at P_x may be topological, granting them may change the size of P_ℓ for some leaves ℓ in $\mathcal{T}_x$, hence the procedure has to ensure that the size of P_x does not become too large after the current requests will be granted. Indeed, if $|P_x|$ is soon to exceed some threshold σ_i that (just like the threshold ρ_i) depends on the level i of x in $\mathcal{T}$, then the execution of Procedure `Update` at x is halted and the procedure is reinvoked at the parent of x in $\mathcal{T}$. Otherwise, the nodes in P_x are reassigned to leaves in $\mathcal{T}_x$ in a manner that keeps $|P_\ell|$ sufficiently small for all leaves ℓ in $\mathcal{T}_x$. The threshold σ_i is designed so that $\sigma_h = 3N_0/2$, thus if $|P|$ exceeds the threshold σ_h, then at least $N_0/2 > M/2$ node insertion requests must have been granted.

The binary tree. Fix $h = \lfloor \log M \rfloor - 1$. Our controller maintains an implicit complete binary tree $\mathcal{T}$ of height h. To avoid confusion with the nodes of the path, the basic elements of the binary tree $\mathcal{T}$ are referred to as *vertices*. Let $\ell_1, \ldots, \ell_{2^h}$ be the *leaves* of $\mathcal{T}$, namely, the vertices in level 0, and let $(u_1, \ldots, u_n)$ be the path P at some stage of the execution ordered from left to right (so that u_1 is the leftmost node and u_n is the rightmost node). The nodes of P are *assigned* to the leaves of $\mathcal{T}$ in a manner preserving the following *monotonicity property*: if u_i is assigned to ℓ_j for some $1 \leq i < n$ and $1 \leq j \leq 2^h$, then u_{i+1} is assigned to one of the leaves $\ell_j, \ell_{j+1}, \ldots, \ell_{2^h}$. Bear in mind that the assignment of nodes to leaves may change occasionally, but the monotonicity property is always preserved. Node u_i may be unaware of its exact rank i in P, but it always knows to which leaf in $\mathcal{T}$ it is assigned.

Given some internal vertex x in $\mathcal{T}$, we say that the node u is *assigned* to x if u is assigned to one of the leaves in the subtree $\mathcal{T}_x$ (that is, the subtree of $\mathcal{T}$ rooted at x). By the monotonicity property, the subset of nodes assigned to the vertex x forms a (contiguous) subpath of P, denoted by P_x. Clearly, the collection $\{P_x \mid x \in \mathcal{T}\}$ forms a laminar family.

The implicit binary tree $\mathcal{T}$ is simulated by the nodes of P. Each vertex x in $\mathcal{T}$ is represented

by the leftmost node in P_x. (The root z is represented by the leftmost node in $P_z = P$.) In what follows, we may sometimes refer to the vertex x itself instead of referring to the node in P that represents x, e.g., we may talk about some action taken by vertex x when we actually mean that the node representing x is taking this action on behalf of x. Let x be some internal vertex and let x_l and x_r be its left and right children in $\mathcal{T}$, respectively. Observe that as x and x_l are represented by the same node in P, exchanging information between them does not cost anything (in terms of message complexity). However, $|P_{x_l}|$ messages should be delivered in order to exchange information between x and x_r.

In a preprocess stage, the controller assigns the N_0 nodes of P to the leaves of $\mathcal{T}$ so that there are between $\lfloor N_0/2^{h-i} \rfloor$ and $\lceil N_0/2^{h-i} \rceil$ nodes assigned to x for every vertex x of level $0 \leq i \leq h$ in $\mathcal{T}$. It also spreads the M permits in P among the leaves of $\mathcal{T}$ so that there are between $\lfloor M/2^{h-i} \rfloor$ and $\lceil M/2^{h-i} \rceil$ permits stored in P_x for every vertex x of level $0 \leq i \leq h$ in $\mathcal{T}$. In particular, each leaf stores between $\lfloor M/2^h \rfloor$ and $\lceil M/2^h \rceil$ permits, where by the choice of h, we have $2 \leq M/2^h < 4$. Throughout the execution permits may be shifted along P, either to be granted to some requests (in which case they are eliminated from the network) or, as we describe soon, in attempt to balance between leaves storing different numbers of permits.

Locked nodes. Every node in the path is always either *locked* or *unlocked*. Intuitively, a node is locked when it participates in handling some requests and it is not available to handle new requests. A request presented at a locked node u will have to "wait for its turn". For that purpose, u maintains a queue denoted by $\mathcal{Q}(u)$ so that each request presented at u while u is locked is enqueued into $\mathcal{Q}(u)$. When u becomes unlocked, it checks whether $\mathcal{Q}(u)$ contains any request; in case it does, the request at the head of $\mathcal{Q}(u)$ is dequeued, its handling procedure (to be described soon) is initiated, and u becomes locked again. As we prove later on, it is guaranteed that a locked node will become unlocked within finite time, hence a request cannot remain in the queue indefinitely.

In fact, we consider $h + 1$ different types of locks: every locked node u maintains a variable $\lambda(u)$ that stores the level $0 \leq i \leq h$ of the last ancestor x of u in $\mathcal{T}$ which is responsible for locking it; u remains locked with $\lambda(u) = i$ until x instructs it to become unlocked, in which case we set $\lambda(u) \leftarrow -\infty$, or until some ancestor y of u of level $i < j \leq h$ in $\mathcal{T}$ takes responsibility for locking u, in which case we set $\lambda(u) \leftarrow j$ (this will be explained soon).

The values of the $\lambda(\cdot)$ variables are increased by sending `lock` messages. Consider some vertex x of level $0 \leq i \leq h$ in $\mathcal{T}$ and let u be some node in P_x. When u receives a `lock`(i) message from its left (respectively, right) neighbor in P_x, if $\lambda(u) < i$, then u sets $\lambda(u) \leftarrow i$ and sends a `lock`(i) message to its right (resp., left) neighbor in P_x (assuming that such a neighbor exists); otherwise ($\lambda(u) \geq i$), u does nothing.

Handling a request. We now turn to describe the request-handling process. As explained above, this process is initiated either when a request is presented at an unlocked node with an empty queue or when a node with a non-empty queue becomes unlocked. Consider some path node u and let ℓ be the leaf of $\mathcal{T}$ to which u is assigned. The first thing u does when it starts handling a request is to lock itself by setting $\lambda(u) \leftarrow 0$ and to send a `lock`(0) message to both its left and right neighbors in P_ℓ. Consequently, the `lock`(0) message propagates to both directions along P_ℓ so that eventually it reaches all nodes in P_ℓ. Note that every node in P_ℓ sends at most two `lock`(0) messages between the last time it became unlocked and the next time it becomes unlocked.

When the rightmost node in P_ℓ receives a `lock`(0) message, it sends an `update`(0) message to ℓ in attempt to invoke Procedure `Update` (to be described soon) at ℓ. Since ℓ is represented by the leftmost node in P_ℓ, it is guaranteed that when this `update`(0) message arrives to ℓ, all nodes in P_ℓ are already locked.

Procedure `Update`. Consider some vertex x of level $0 \leq i \leq h$ in $\mathcal{T}$ upon receiving an `update`(i) message at time t. This message is ignored if $\lambda(x) > i$ or if $\lambda(x) = i > 0$. Otherwise, Procedure `Update` is invoked at x. In what follows, we denote by A_x^t the set of requests that wait for permits at the nodes of P_x at time t (excluding the requests at the queues).

The first thing that Procedure `Update` does is to set $\lambda(x) \leftarrow i$ and to propagate a `lock`(i) message from left to right along P_x. (Actually, this is not necessary if $i = 0$, since all nodes in P_ℓ must be locked when Procedure `Update` is invoked at the leaf ℓ.) Next, the procedure gathers to x all permits that are currently stored throughout P_x, denote their number by $p(x)$. The procedure also calculates the size $r(x)$ of A_x^t and the predicted size $s(x)$ of P_x once the topological requests in A_x^t are granted and executed, that is, $s(x)$ is the current size of P_x plus the number of node insertion requests in A_x^t minus the number of node deletion requests in A_x^t.

Gathering the permits to x and calculating $p(x)$, $r(x)$, and $s(x)$ is performed by propagating a single message from left to right along P_x and then propagating a single message from right to left along P_x. This process, referred to as the *counting propagations* of Procedure `Update`, ends when the right-to-left propagation returns to x at some time $t' > t$. It may be the case that by time t', the value of $\lambda(x)$ has increased and exceeded i because of a concurrent invocation of Procedure `Update` at some ancestor of x in $\mathcal{T}$. If this indeed happens, then x ignores the return of the counting propagations at time t' which essentially causes the current invocation of Procedure `Update` to halt. Otherwise, we still have $\lambda(x) = i$ at time t' and x continues as follows.

Fix $\phi = 1 + 1/h$ and $H = \frac{\log N_0 + \log 3 - 1}{1 - \log \phi}$. For $0 \leq i \leq h$, define $\rho_i = \frac{M/2}{(2\phi)^{h-i}}$ and $\sigma_i = (2/\phi)^{H-h+i}$. Observe that $\rho_h = M/2$ and $\sigma_h = 3N_0/2$, while $\Omega(1) \leq \rho_0 < 1$ and $\sigma_0 = O(N_0/2^h)$. Now, x checks whether

$$p(x) - r(x) > \rho_i \quad \text{and} \quad s(x) < \sigma_i \ . \tag{1}$$

If both inequalities in (1) hold, then Procedure `Grant`, to be described soon, is invoked at x. So,

assume that at least one of the inequalities in (1) does not hold. If $i < h$, namely, if x is not the root of $\mathcal{T}$, then x sends an `update`$(i+1)$ message to its parent in $\mathcal{T}$. Otherwise (x is the root of $\mathcal{T}$), Procedure `Reject`, to be described soon, is invoked at x.

Procedure `Grant`. Consider some descendant y of x of level $0 \leq j \leq i$ in $\mathcal{T}$. The goals of Procedure `Grant` are (1) to grant permits to the requests in A_x^t; (2) to spread the $p(x) - r(x)$ remaining permits among the leaves of $\mathcal{T}_x$ so that the number of permits stored in P_y is between $\lfloor (p(x) - r(x))/2^{i-j} \rfloor$ and $\lceil (p(x) - r(x))/2^{i-j} \rceil$; (3) to reassign the nodes in P_x to the leaves of $\mathcal{T}_x$ so that the number of nodes in P_y is between $\lfloor s(x)/2^{i-j} \rfloor$ and $\lceil s(x)/2^{i-j} \rceil$; and (4) to unlock the nodes in P_x. This is performed by propagating a single message from left to right along P_x. This message essentially encodes the value of $p(x) - r(x)$, the value of $s(x)$, and the rank j of the current node u_j in what is soon to become $P_x = (u_1, \ldots, u_{s(x)})$. We think of the preprocess stage of the controller as an invocation of Procedure `Grant` at the root of $\mathcal{T}$.

Procedure `Reject`. Procedure `Reject` is always invoked at the root $z = x$ of $\mathcal{T}$. The goals of the procedure are (1) to grant permits to $\min\{p(z), r(z)\}$ of the requests in A_z^t (say, the leftmost); (2) to reject all other requests in A_z^t; (3) to unlock the nodes in P; and (4) to inform all nodes in P that they should reject any subsequent request. This is performed by propagating a single message from left to right along P.

Correctness. We would like to show that every request is handled (either granted a permit or rejected) within finite time and that if a request is rejected, then eventually, at most $M/2$ permits remain in P. We start with the former. First, by the definition of the request handling process, we have the following invariant.

Invariant 3.2. *Consider some leaf ℓ in $\mathcal{T}$ and let u be some node in P_ℓ at time t. If $\lambda(u) \leftarrow 0$ at time t, then there exists some $t < t' < \infty$ such that ℓ receives an* `update`(0) *message at time t'.*

Next, consider some vertex x of level $0 \leq i \leq h$ in $\mathcal{T}$ and suppose that the counting propagations of Procedure `Update` invoked at x at time t return to x at time $t' > t$. If $\lambda(x) = i$ at time t', which means that Procedure `Grant`/`Reject` is about to be invoked, then $\lambda(v) = i$ at time t' for all nodes $v \in P_x$. Therefore since `lock`(k) messages, $k > i$, are always propagated from left to right, the FIFO channel assumption guarantees that the execution of Procedure `Grant`/`Reject` is not "disturbed" by such `lock`(k) messages and can be considered as an atomic operation. (Actually, this is obvious for Procedure `Reject` which is always invoked at level h.) In particular, we have the following invariant.

Invariant 3.3. *Consider some vertex x of level $0 \leq i \leq h$ in $\mathcal{T}$ and let u be some node in P_x at time t. If Procedure* `Grant` *or Procedure* `Reject` *are invoked at x at time t, then there exists some $t < t' < \infty$ such that $\lambda(u) \leftarrow -\infty$ at time t'.*

We are now ready to establish the following lemma.

Lemma 3.4. *Let u be some node in P and consider some level $0 \leq i \leq h$. If $\lambda(u) \leftarrow i$ at time t, then there exists some $t < t' < \infty$ such that $\lambda(u) \leftarrow -\infty$ at time t'.*

Proof. Consider the vertex x of level i in $\mathcal{T}$ such that $u \in P_x$ at time t. We will show that either there exists some $t < t' < \infty$ such that $u \in P_x$ and Procedure `Grant/Reject` is invoked at x at time t' or there exists some $t < t' < \infty$ such that $\lambda(u) \leftarrow j > i$ at time t'. The assertion follows by Invariant 3.3 and since the value of $\lambda(u)$ can increase at most h times before it is set to $-\infty$.

Suppose first that $i > 0$. The variable $\lambda(u)$ was set to i at time t because of some left-to-right propagation of `lock`(i) messages along P_x that started at x at time $t_1 \leq t$. This propagation was immediately followed by the counting propagations of Procedure `Update` — denote by $t_2 > t$ the time at which they returned to x. We know that $\lambda(x) = i$ at time t_1. If $\lambda(x)$ remained unchanged until time t_2, then Procedure `Grant` or Procedure `Reject` were invoked at x at time t_2 and the assertion holds. Otherwise, it must be that $\lambda(x) \leftarrow j > i$ at time $t_1 < t_3 < t_2$. Consequently, `lock`(j) messages were propagated to the right from x. By the FIFO channel assumption, this propagation must have reached u and set $\lambda(u) \leftarrow j$ after time t.

Now suppose that $i = 0$. Invariant 3.2 guarantees that there exists some $t_4 > t$ such that x receives an `update`(0) message at time t_4. If $\lambda(x) = 0$ at time t_4, then Procedure `Update` is invoked at x at time t_4 and the proof is analogous to the case $i > 0$. Otherwise, it must be that $\lambda(x) \leftarrow j > 0$ at time $t_5 < t_4$. Take t_5 to be the latest such time. A propagation of `lock`(j) to the right started at x at time t_5. The point is that this propagation must have reached u and set $\lambda(u) \leftarrow j$ after time t as otherwise, we would not have $\lambda(u) \leftarrow 0$ at time t. □

Lemma 3.4 immediately implies that every request is handled within finite time. It remains to show that if a request is rejected, then eventually, at most $M/2$ permits remain in P. Recall that the controller does not reject any request unless Procedure `Reject` is invoked at the root z of $\mathcal{T}$ at some time t. This procedure is invoked when one of the inequalities in (1) does not hold for z. If the left inequality does not hold, then the number of permits that remain in P if all requests in A_z^t are granted is at most $\rho_h = M/2$. If the right inequality does not hold, then the size of P if all requests in A_z^t are granted is at least $\sigma_h = 3N_0/2 > N_0 + M/2$ which means that in total, more then $M/2$ node insertion requests must have been granted. Therefore, in both cases it is guaranteed that upon completion of Procedure `Reject`, at most $M/2$ permits remain in P.

Analysis. We now turn to establish an $O(N \log^2 M)$ upper bound on the message complexity of our controller. This is done by proving that the total number of messages sent throughout the execution is $O(N_0 h^2)$. The controller sends $O(N_0)$ messages in the preprocess stage in order to spread the M permits among the 2^h leaves of $\mathcal{T}$, so in what follows we consider the messages sent during the actual execution of the controller. We first argue that it is sufficient to bound the number

of `lock`(i) messages, $0 \leq i \leq h$. Indeed, for every $0 \leq i \leq h$, the `lock`(i) messages asymptotically dominate the `update`(i) messages, the messages sent during the counting propagations in level i, and the messages sent during the executions of Procedure `Grant`/`Reject` in level i.

Consider some node u in P. Every time u sets $\lambda(u) \leftarrow i$ accounts for at most two `lock`(i) messages sent to u if $i = 0$ and for at most one `lock`(i) message sent to u if $i > 0$. Therefore instead of bounding the number of `lock`(i) messages sent to u, we shall bound the number of setting $\lambda(u) \leftarrow i$ operations. We already know that if u sets $\lambda(u) \leftarrow i$, then eventually u becomes unlocked (and sets $\lambda \leftarrow -\infty$) either by Procedure `Grant` or by Procedure `Reject` invoked at some vertex of level $i \leq j \leq h$ in $\mathcal{T}$ to which u is assigned. We shall partition the setting $\lambda(u) \leftarrow i$ operations to those *eventually unlocked* by Procedure `Grant` and to those *eventually unlocked* by Procedure `Reject`. Every node is unlocked by Procedure `Reject` exactly once, thus summing over all nodes and all levels, we conclude that the total number of setting $\lambda(\cdot) \leftarrow i \in \{0, \ldots, h\}$ operations which are eventually unlocked by Procedure `Reject` is $O(N_0 h)$. In what follows we focus on the setting $\lambda(\cdot) \leftarrow i \in \{0, \ldots, h\}$ operations which are eventually unlocked by Procedure `Grant`.

Consider some vertex x of level $0 < i \leq h$ and some descendant y of x of level $0 \leq j \leq i$ in $\mathcal{T}$. Let I be some invocation of Procedure `Grant` at x. We denote by $t_y(I)$ the time at which the left-to-right propagation of I along P_x reaches the last node which is (re)assigned to y. We argue that the size of P_y at time $t_y(I)$ is $O(N_0/2^{h-j})$. Indeed, when Procedure `Grant` is invoked at x, we have $s(x) < \sigma_i$, therefore the procedure assigns less than $\lceil \sigma_i / 2^{i-j} \rceil = O(N_0/2^{h-j})$ nodes to y. In particular, for every leaf ℓ of $\mathcal{T}_x$, we have $|P_\ell| = O(N_0/2^h)$ at time $t_\ell(I)$.

Now, a permit granted to a request presented at some node u accounts for at most $|P_\ell|$ setting $\lambda(\cdot) \leftarrow 0$ operations, where ℓ is the leaf in $\mathcal{T}$ to which u is assigned. Therefore the total number of setting $\lambda(\cdot) \leftarrow 0$ operations which are eventually unlocked by Procedure `Grant` is $O(M \cdot N_0/2^h) = O(N_0)$. Consider some vertex x of level $0 < i \leq h$ in $\mathcal{T}$. A node $u \in P_x$ sets $\lambda(u) \leftarrow i$ only as a result of invoking Procedure `Update` at x and each such invocation accounts for $|P_x| = O(N_0/2^{h-i})$ setting $\lambda(\cdot) \leftarrow i$ operations. Therefore in order to bound the total number of setting $\lambda(\cdot) \leftarrow i$ operations which are eventually unlocked by Procedure `Grant`, we will bound the total number of invocations of Procedure `Update` at level i vertices which are eventually unlocked by Procedure `Grant`.

Let us focus on one such invocation of Procedure `Update` at x. This invocation is due to an `update`(i) message sent to x from one of its children w in $\mathcal{T}$ when w found out at some time t that one of the inequalities in (1) is violated, namely, that either $p(w) - r(w) \leq \rho_{i-1}$ or $s(w) \geq \sigma_{i-1}$. Let I_1 (respectively, I_2) denote the latest (resp., earliest) invocation of Procedure `Grant` at an ancestor y_1 (resp., y_2) of x of level $i \leq j_1 \leq h$ (resp., of level $i \leq j_2 \leq h$) in $\mathcal{T}$ at time $t_1 < t$ (resp., at time $t_2 > t$). (Invocation I_1 exists by definition. Invocation I_2 exists since we assumed that the corresponding locked vertices are eventually unlocked by Procedure `Grant`.)

Lemma 3.5. *Let R be the set of requests in P_w which are granted permits during the time interval $(t_w(I_1), t_w(I_2)]$. Then $|R| = \Omega\left(\frac{M}{h 2^{h-i}}\right)$.*

We will soon establish Lemma 3.5, but first note that it implies that throughout the execution such `update`(i) messages cannot be sent to level i vertices in $\mathcal{T}$ more than $h2^{h-i}$ times. Therefore the total number of setting $\lambda(\cdot) \leftarrow i$ operations which are eventually unlocked by Procedure `Grant` is $O(h2^{h-i} \cdot N_0/2^{h-i}) = O(N_0 h)$. Summing over all i levels, we conclude that the total number of setting $\lambda(\cdot) \leftarrow i \in \{1, \ldots, h\}$ operations which are eventually unlocked by Procedure `Grant` is $O(N_0 h^2)$. It remains to establish Lemma 3.5.

Proof of Lemma 3.5. At time $t_w(I_1)$, there were less than

$$\left\lceil \sigma_{j_1}/2^{j_1-i+1} \right\rceil = \left\lceil \frac{2^{H-h+i-1}}{\phi^{H-h+j_1}} \right\rceil = \left\lceil \sigma_{i-1}/\phi^{j_1-i+1} \right\rceil \leq \left\lceil \sigma_{i-1}/(1+1/h) \right\rceil$$

nodes assigned to w and more than

$$\left\lfloor \rho_{j_1}/2^{j_1-i+1} \right\rfloor = \left\lfloor \frac{M/2}{2^{h-i+1} \cdot \phi^{h-j_1}} \right\rfloor = \left\lfloor \rho_{i-1} \cdot \phi^{j_1-i+1} \right\rfloor \geq \left\lfloor \rho_{i-1} \cdot (1+1/h) \right\rfloor$$

permits stored in P_w. We know that if the requests in $A_w^t \subseteq R$ are to be granted permits by w at time t, then either at most ρ_{i-1} permits will remain in P_w or the size of P_w will become at least σ_{i-1}. Therefore $|R|$ is greater than

$$\min\left\{\rho_{i-1} \cdot (1+1/h) - \rho_{i-1}, \sigma_{i-1} - \sigma_{i-1}/(1+1/h)\right\} = \min\left\{\frac{\rho_{i-1}}{h}, \frac{\sigma_{i-1}}{h+1}\right\} .$$

The assertion follows since $\rho_{i-1} = \Omega(M/2^{h-i})$ and $\sigma = \Omega(N_0/2^{h-i})$ (recall our assumption that $M < N_0$). □

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] Y. Afek and M. Ricklin. Sparser: a paradigm for running distributed algorithms. *J. Algorithms*, 14(2):316–328, 1993.

[3] Y. Afek and M.E. Saks. Detecting global termination conditions in the face of uncertainty. In *Proc. 7th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 109–124, 1987.

[4] A. Andersson and T. W. Lai. Fast updating of well-balanced trees. In *Proc. 2nd Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 111–121, 1990.

[5] B. Awerbuch, S. Kutten, and D. Peleg. Competitive distributed job scheduling (Extended Abstract). In *Proc. 24th ACM Symp. on Theory of Computing (STOC)*, pages 571–580, 1992.

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

[7] M.A. Bender, R. Cole, E.D. Demaine, M. Farach-Colton and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proc. 10th Ann. European Symp. on Algorithms (ESA)*, pages 152–164, 2002.

[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[9] P. F. Dietz. Maintaining Order in a Linked List. In *Proc. 14th ACM Symp. on Theory of Computing (STOC)*, pages 122–127, 1982.

[10] P. F. Dietz, J. I. Seiferas, and J. Zhang. A tight lower bound for online monotonic list labeling. *SIAM J. Discrete Math.* 18(3):626–637, 2004.

[11] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th ACM Symp. on Theory of Computing (STOC)*, pages 365–372, 1987.

[12] P. F. Dietz and J. Zhang. Lower bounds for monotonic list labeling. In *Proc. 2nd Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 173–180, 1990.

[13] M.J. Fischer, N.A. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[14] A. Itai, A. Konheim, and M. Rodeh. A sparse table implementation of priority queues. In *Proc. 8th Colloq. on Automata, Languages and Programming (ICALP)*, pages 417–431, 1981.

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

[16] A. Korman. Improved compact routing schemes for dynamic trees. In *Proc. 27th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 185–194, 2008.

[17] A. Korman and S. Kutten. Controller and estimator for dynamic networks. In *Proc. 26th ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing (PODC)*, pages 175–184, 2007.

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

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

[20] C. Lund, N. Reingold, J. Westbrook, and D.C.K. Yan. Competitive on-line algorithms for distributed data management. *SIAM J. Comput.*, 28(3):1086–1111, 1999.

[21] S. Kutten. Optimal fault-tolerant distributed construction of a spanning forest. *Inf. Process. Lett.*, 27(6):299–307, 1988.

[22] A. K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Inform.*, 21:101–112, 1984.

[23] D. Willard. Maintaining dense sequential files in a dynamic environment. In *Proc. 14th ACM Symp. on Theory of Computing (STOC)*, pages 114–121, 1982.

APPENDIX

A A reduction from trees to paths — the dynamic case

In this section we sketch the outlines of the the adaptation of the reduction presented in Section 3.1 to the dynamic case, that is, we explain how an $(M, M/2)$-controller that operates on a tree T and supports topological and non-topological requests can be implemented by an $(M, M/2)$-controller that operates on a path P and supports topological and non-topological requests. We first argue that we may assume without loss of generality that the requests presented at some tree node u are handled on a first-in-first-out basis such that a request is never presented before the previous request was granted/rejected. To see why this is a valid assumption note that the controller may delay the handling process of a request until the handling processes of all previous requests are over. Assuming that the handling process of each request takes finite time, it follows that no request is delayed indefinitely. Therefore, we may as well assume that the next request is presented at u after the current request is granted/rejected.

Recall that in the reduction described in Section 3.1, each tree node u was associated with two path nodes: $d[u]$ and $f[u]$. In the dynamic case, each tree node is associated with at most three path nodes according to the following principal: at any given time the tree node u is associated with at least one of the path nodes $d[u], f[u]$; it may also be associated with the path node $\mathrm{tmp}[u]$. As before, each node in P has a unique preimage in T. The role of the path nodes $d[u]$ and $f[u]$ is the same as in Section 3.1. The path node $\mathrm{tmp}[u]$ is introduced in order to support node insertion requests as we shall describe soon. In particular, at all times we have $|P| \leq |T| \leq 3|P|$.

The implementation of the $(M, M/2)$-controller on T is based on invoking the path controller in iterations, where in the i^{th} iteration an $(M_i, M_i/2)$-controller is invoked on a path P (rooted at its leftmost node), initially consisting of the nodes $d[u]$ and $f[u]$ for every tree node u. We will soon show that the total number of iterations is $O(1)$. The parameter M_i is the number of permits that remain in T at the beginning of the iteration, so $M_1 = M$. The i^{th} iteration ends when the path controller rejects some request. When this happens, the tree controller grants permits to the currently awaiting requests and counts the number M_{i+1} of remaining permits. If $M_{i+1} \leq M/2$, then the execution of the tree controller is over and it rejects all subsequent requests automatically. Otherwise ($M_{i+1} > M/2$), the $(i+1)^{\text{st}}$ iteration starts. This process is implemented by sending $O(1)$ messages along each tree link, and hence it incurs an average message complexity of $O(1)$ (on top of the average message complexity of the path controllers). We now turn to describe the implementation of the tree controller during the i^{th} iteration.

In contrast to the translation described in Section 3.1, a request presented at some tree node u may now correspond to one or two requests in the path controller. A non-topological request presented at u is translated to one non-topological request presented either at $d[u]$ or at $f[u]$ (recall

that at least one of them exists). As before, if the path controller grants a permit to this request, then the request at u is granted a permit by the tree controller. The case of topological requests is slightly more involved.

Consider first a request presented at u for the insertion of some child v of u and let $d[v]$ and $f[v]$ be the DFS time stamps of v in the tree T' resulting from T by the insertion of v. In response to this request, u first presents a child insertion request at the path node $d[v]-1$ (whose preimage is either a child of u or u itself). If the request at $d[v]-1$ is granted a permit by the path controller, then (1) a new node is inserted into P — denote it $\mathrm{tmp}[u]$ and associate it with u for the time being; and (2) u presents a child insertion request (another one) at the path node $f[v]-1$ (whose preimage is either a child of v or v itself in T', and hence either a child of u or u itself in T). If the request at $f[v]-1$ is also granted a permit by the path controller, then the tree controller grants a permit to the request at u and v is inserted into T. When this happens, $\mathrm{tmp}[u]$ is renamed $d[v]$ and it becomes associated to v.

An issue that should be discussed is the permit distribution. In the static case, the permit distribution among the nodes of P determined at any given time the permit distribution among the nodes of T in a straightforward manner. Things now are slightly more complicated because when the path controller grants a permit to the request at $d[v]-1$, the tree controller still does not grant any permit. Instead, the tree controller *freezes* one permit at the preimage of $d[v]-1$. This frozen permit will not be used until the end of the iteration. When the path controller grants a permit to the request at $f[v]-1$, we deliver one permit from the preimage of $f[v]-1$ to the preimage of u and grant this permit to the request presented at u.

Now, consider a request presented at u for the deletion of some child v of u. In response to this request, u first presents a child deletion request at the path node $f[v]-1$ (whose preimage is either a child of v or v itself, and hence either a grandchild or a child of u). If the request at $f[v]-1$ is granted a permit by the path controller, then (1) the node $f[v]$ is deleted from P; and (2) u presents a child deletion request (another one) at the path node $d[v]-1$ (whose preimage is either a child of u or u itself). If the request at $d[v]-1$ is also granted a permit by the path controller, then the tree controller grants a permit to the request at u and v is deleted from T.

Similarly to the implementation of the node insertion request, we will have to take care of the permit distribution: When the path controller grants a permit to the request at $f[v]-1$, the tree controller freezes one permit at the preimage of $f[v]-1$. When the path controller grants a permit to the request at $d[v]-1$, we deliver one permit from the preimage of $d[v]-1$ to the preimage of u and grant this permit to the request presented at u.

It remains to show that the number of iterations of the path controller is indeed $O(1)$. In the i^{th} iteration the path $(M_i, M_i/2)$-controller is invoked, where $M_i > M/2$ is the number of permits remaining in T. The iteration ends when this path controller rejects some requests. By definition, this does not happen unless at least $M_i/2$ requests were already presented at the nodes of P, which

corresponds to at least $M_i/4 = M/8$ requests presented at the nodes of T (recall that each tree request is translated to at most two path requests). When the iteration ends, all this requests are granted, therefore each iteration accounts for granting at least $M/8$ tree requests. It follows that in total, there are at most 8 iterations.