Controller and Estimator for Dynamic Networks

Amos Korman* and Shay Kutten†
Faculty of IE&M, Technion
Haifa, Israel
pandit@tx.technion.ac.il, kutten@ie.technion.ac.il

ABSTRACT

Afek, Awerbuch, 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. This counting problem 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: 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).

An efficient Resource Controller was constructed in the paper by Afek et al., which can operate on a dynamic network assuming that the network is spanned by a tree that may only grow, and only by allowing leaves to join the tree. In contrast, the Resource Controller presented here can operate under a more general dynamic model, allowing the spanning tree of the network to undergo both insertions and deletions of both leaves and 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 controller of Afek et al. 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 an efficient *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. Informally, the resulted new size-estimation protocol uses $O(\log^2 n)$ amortized message complexity per topological change (assuming that the number of changes in the network size is "not too small"), where n is the current number of nodes in the network. In addition, with the same message complexity (as that of the size-estimation protocol), the new controller can be used to solve the *name-assignment* problem by assigning and maintaining unique $\log n + O(1)$-bit identifiers for the nodes of the dynamic network.

The new size-estimation protocol can be used for other applications, not mentioned in the paper by Afek et al.. Specifically, it 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 more general models. These extensions maintain the same asymptotic sizes of the corresponding labels (or routing tables) of the original schemes and incur only a relatively low extra additive cost to the message complexity of the corresponding original schemes.

Categories and Subject Descriptors: F.2.2: Analysis oF Algorithms and Problem Complexity, C.2.2: Network Protocols.

General Terms: Algorithms.

Keywords: Distributed algorithms, Labeling schemes, Asynchronous protocols.

1. INTRODUCTION

In common sequential settings, an online algorithm (notably, a competitive algorithm, such as those in [25]) 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, namely, 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 a 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

*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.
†Supported in part by a grant from the Israel Science Foundation.

PODC'07, August 12–15, 2007, Portland, Oregon, USA.

shortest path algorithm [10] on the graph represented in the node's own memory. This approach is the one used in the Internet, See, e.g., the OSPF protocol [26].

This paper addresses problems affected by both kinds of uncertainties. The *token collection* problem [3] is a variant of the updating problem: count (at one center node) the (approximate) number of events that occur all over the network. Another variant is the *size-estimation* problem, in which it is required to maintain (at a center node) a constant approximation to the number of nodes in the dynamic networks. It was shown in [1] that these two (and several other) problems can be reduces to the following (M, W)-*Controller* problem: when an event "wishes to occur" at some node, the algorithm has to move either a *permit* or a *reject* from some center to the node requesting the event. If the requesting node receives a permit, the permit is consumed at the node and subsequently, the event occurs. 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). As shown in [1], the token collection and size-estimation problems can be reduced to the (M, W)-*Controller* problems, even if the number of events (or topology changes) is not bounded. Intuitively, the reduction operates in iterations. When the controller needs to give a reject, it refrains from giving it and instead, a new iteration of the controller is initiated with a larger M.

Note, that the message 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 message 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 offline sequential algorithm whose message complexity is $O(n)$.

The controller of [1] was designed to work under the *controlled* dynamic 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 (we discuss this subject in Subsection 1.1). In [1], it was also assumed that the network is spanned by a tree that may only grow, and only by allowing leaves to join the tree. I.e., only one type of topology changes is allowed: an insertion of a leaf node. 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 resource controller presented here (still in the controlled model) can be applied under the more general dynamic model, allowing the spanning tree of the network to undergo both insertions and deletions of leaves as well as 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.

Intuitively, it is not clear how to adapt the previous controller of [1] to handle these additional topology changes efficiently. 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 strongly on its precise distance to the root. The type of topology change considered in [1] (the insertion of a leaf node) 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 away 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 controller of [1], our (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.

1.1 Motivating the model

As in [1], we too assume the *controlled dynamic* model, in which a topological change does not happen instantaneously. Instead, it is delayed until getting a permit to do so from the resource controller. (See the model, Section 2.1.) Such a model may be found useful e.g. in overlay networks. Consider, for example, a Peer to Peer network (P2P) which is dedicated to users who are interested in a certain subject. When a user becomes non-interested in the subject, its node leaves the network in a graceful manner. Similarly, when a node becomes interested in the subject, its node joins the network in a graceful manner. In this context, it seems reasonable that the change can be delayed further (beyond the inherent delay), so that our controller could be applied.

One can use, for example, a controller at a layer above the overlay network (and below the application of that network). This layer would present to the application a more orderly overlay network, one for which the number of nodes is known (and can be controlled), nodes are labeled economically to support an efficient routing scheme, some queries can be answered (e.g., a query about the lowest common ancestor of given two nodes in a tree), etc.

As mentioned, this paper does not handle spontaneous crashes. However, this seems to be less crucial in the case of overlay networks, where most of the topology changes are those decided upon by designers or by algorithms, as opposed to changed caused by uncontrollable faults. (One reason that in overlay networks there are more controlled changes than uncontrolled ones is that, in overlay networks, the cost of performing a desired change is relatively low, compared to traditional networks; this has been motivating

many intentional changes; on the other hand, the reliability of networks in general has increased, making spontaneous faults more rare.)

From the theoretical point of view, the controlled dynamic 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. [17, 20, 21]). On the other extreme, lies the "chaotic" fully asynchronous and adversarial model. The middle ground controlled model has also a theoretical appeal, 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.

1.2 Other 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 [9, 25]. 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 moving a server 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 k-server problem and the controller problem is that in the latter, 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 appear in [5, 8, 4] 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 [23] 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*. The complexity of the majority commitment protocol of [23] is $O(n^2 \log n)$ messages (each of $O(\log n)$ bits). The majority commitment problem in such a setting (of initial faults in an asynchronous network) was previously presented in the famous paper of Fischer, Lynch, and Paterson [11] and solved there 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 [6] and an $O(n \log^5 n)$ solution appears in [3]. The solution for the majority commitment problem in [3] was solved using a size estimation protocol for growing trees which uses $O(n \log^4 n)$ messages (in the terms used here).

The granting of permits to requests was studied in [24] in the case that permits are distributed in the network nodes originally, according to some probability distribution.

The problem of dynamically maintaining routing schemes and other informative labeling scheme representations was studied in [2, 17, 20, 21, 22].

1.3 Our contributions

In this paper, we consider dynamic networks which are spanned by a spanning tree that may undergo both additions and deletions of both leaves and internal nodes. For such dynamic networks, we establish (M, W)-controllers which are efficient in terms of their message complexity.

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 sequential 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 sequential 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 controller 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 [1] (however, we assume that each message is encoded using $O(\log n)$ bits while the controller in [1] assumes that each message contains only $O(\log \log n)$ bits).

In addition, under the more general dynamic model, our controller can handle all the applications mentioned in [1] using the above mentioned message complexity. In particular, under the more general dynamic model, the new controller solves the size-estimation problem using message complexity $O(n_0 \log^2 n_0) + O(\sum_j \log^2 n_j)$. The same message complexity is used to solve the name-assignment problem by assigning and 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 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 [1]. We extend many existing labeling schemes supporting different queries (e.g routing, ancestry, etc.) so that these schemes can now operate correctly also under more general models. These extensions maintain the same asymptotic sizes of the corresponding labels (or routing tables) of the original schemes and incur only a relatively low extra additive cost to the message complexity of the corresponding original schemes. For example, we extend the ancestry labeling schemes on trees mentioned in [14, 17, 20] to support also controlled deletions of both leaves and internal nodes, and we extend the routing labeling schemes on trees mentioned in in [2, 12] to support also controlled deletions of leaves. All these extensions use the same asymptotic sizes of labels and incur only an extra additive cost of $O(n_0 \log^2 n_0) + O(\sum_j \log^2 n_j)$ beyond the message complexity of the corresponding original schemes. (See Section 5 for more details.)

1.4 Paper outline

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 descriptions 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 for analyzing 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 full paper. The discussion regarding the applications of the new controller appears in Section 5. Section 6 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 (V, E), 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. 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.

Given a tree T, 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.

We assume that the (dynamic) network is spanned by a spanning tree T that may undergo the following types of topological changes.

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 a child of v in the spanning tree T.

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

Add internal node (between neighbors v and w in T): Edge $e = (v, w)$ splits into two edges (v, u) and (u, w) for a new node u. If v was w's parent, then u is considered 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.

Since all the messages sent by our controller are sent on the edges of the spanning tree T, our controller (and also the controller of [1]) is not affected by additions and removals of non-tree edges, and therefore, such topology changes are also allowed. In addition, we also allow to insert an internal node u inside a non-tree edge (v, w), since this topology change can be considered as composed of adding u as a leaf of v and then adding the non-tree edge (u, w).

Recall, that as in [1], we assume the *controlled* model in which the topology changes do not occur spontaneously. Instead, when an entity wishes to cause an event (including a topology change) at some node u, it enters a *request* at u, and performs the event only after the request is granted a permit from the controller. 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. When granted the permit, the requesting entity in the requesting node u is assumed to perform the topological change in a "graceful" 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.

Incoming and outcoming links from every node are identified by so called *port-numbers*. It is maintained throughout the dynamic scenario, that at any given moment, the port numbers at each vertex v are distinct. We assume that at any time during the execution, each node knows the port number leading to its parent in the spanning tree T. When a new edge is attached to a node v, the corresponding port at v is assigned a unique (among v's ports) port-number. To make our results applicable for a wide range of models, we assume the relatively wasteful model in which the port numbers are assigned by an adversary [12, 17]. We note that the sizes of the port numbers affect the memory complexity of our algorithms, however, assuming these numbers are relatively small ($O(\log n)$ bits per link), our algorithm uses relatively small memory per node (see Section 6).

2.2 Problem Definition

The input of a controller arrives online in the form of *requests* arriving at various nodes. Informally, in order to be useful, the controller should not issue more permits than some given amount M. One the other hand, it should issue some minimum amount of permits before it denies a request for the first time. In fact, since our setting is asynchronous, and non-negligible time may pass between requesting and receiving a permit, it is sufficient to require that if some request is rejected then at least $M - W$ permits are eventually issued to requests, even if these permits are physically issued after the reject takes place. The W parameter is essential the maximum number of 'waisted' permits.

Let us now describe the (M, W)-controller more formally. Initially, an (M, W)-controller in a spanning tree T has a set of M *permits*, and an infinite set of *rejects*, both considered to reside at the tree root. 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 (a permit or a reject) is one of those that originally resided at the root and is currently residing in u. The delivery 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 $M - W$.

Let us note that a controller may control and count any type of event, (e.g., sales of tickets by different nodes, or even the number of messages sent by some other protocol [1]) and not just topology changes. The description in this paper emphasizes the measures the controller uses to handle topological changes. This is because this is the most challenging task. Moreover, any algorithm to control other types of events (non topological) in a dynamic network would have to deal with the topological changes and the dynamic nature

of the network. For example, consider again the case that a protocol sends a message from some node u to some node v. If an adversary can insert internal nodes in an uncontrolled way, the message may never arrive to its destination. Hence, even a controller for a non-topology related events must somehow deal with (and possibly control) the topological changes. The (M, W)-controller described in this paper deals with all types of arriving requests together. That is, the Correctness conditions described above, control and bound the total number of granted requests, no matter to which type of events they correspond.

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 controller of [1], our 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 (whose distance from u is also roughly x), 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 splits 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, throughout the dynamic scenario, 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 holds, 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. The domains satisfy the following invariants. (1) The size of P's domain is about the size of P, and (2) the domains of two packages of the same level are disjoint.

These invariants guarantee that at any time, the number of of permits 'stuck' in 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 granted to requests eventually.

Let us hint how we ensure that domains have the above mentioned properties. Recall, that when a 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 some 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 moves and splits the package P into smaller level packages recursively. In particular, 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 so that the above mentioned domain invariants are maintained. Updating of domains is done only for the sake of analysis, therefore, in particular, the algorithm does not need to use any communication for updating and does not need to notify a nodes about their domain memberships.

3. A NON-DISTRIBUTED CONTROLLER

In this section, we present the new (M, W)-controller in the setting that is similar to that of the sequential k-server problem. In the following section, we show how to implement it distributively, using the non-distributed controller as a high level description of the distributed one. The move complexity used here will translate later into the message complexity in the distributed setting.

3.1 The algorithm

Let us first assume that there exists a fixed and known upper bound U on the number of nodes 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. A node may store multiple packages. 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 contains infinite number of rejects. (Rejects are identical to each other, so a reject package can be represented by a constant number of bits.) 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 precisely $2^i\phi$ 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. It will follow from the description of the algorithm that the level of a (mobile) package is between zero and $\log U + 1$.

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

(1) The creation of either a mobile or a reject package P residing at the root.
- If a mobile package is created then it is created together with a size (which determines a level). This operation increments a variable called **Issued** by the size of P. Variable **Issued** (initially zero) is kept at the root, and is used to count the number of permits issued.
- If a reject package is created, it contains 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 level zero package and the other is a static package containing ϕ permits.
- When a reject package splits, it splits into two reject packages (each containing infinitely many rejects).
(3) The *move* of a package from its host node to a new host 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 $\psi = 4\lceil \log(U)+2 \rceil \cdot \max\{\lceil \frac{U}{W} \rceil, 1\}$. 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, if $j \neq 0$, then $2^j\psi < d(u,w) \leq 2^{j+1}\psi$.

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 one or more 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 (in the tree T). 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 $j(u)$ 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 moving.

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

4. For each $k \in \{0, 1, 2, \cdots, j(u)-1\}$, let u_k be the ancestor of u satisfying $d(u,u_k) = 3 \cdot 2^{k-1}\psi$. Apply the following procedure recursively such that initially, $P = P(u)$.
   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. Every existing mobile package P is associated with a set of (not necessarily existing) nodes, called the *domain* of package P, which may change from time to time. The domains are used for analysis purposes only, therefore, in particular, when a node joins or leaves some domain, no actions are required by the algorithm to support the change. 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) At any time, the currently existing nodes in the domain of each mobile package form a path hanging down (away from the root) from some child of the node holding the package.

Below, we define the domains and show that the domain invariants hold. Initially, there are no packages and no domains. When a package is canceled or becomes static, its domain is canceled. Similarly, when a package splits, its domain is canceled and new domains are given to the new packages that result from the split. A package is *formed* at some time t if, at time t, the package is either created (at the root) or results from a split. (Note, that this happens only after a request arrives at some node u). We first define the domain of a newly formed mobile package.
Let $\rho(u)$ be the vertex defined in Item 3 of the description above. Consider the following cases.
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 PROC($P(u)$) is completed, we have the following. For $k \in \{0, 1, 2, \cdots, j(u) - 1\}$, 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 $Dom(P_k)$ associated with the package P_k is the set of vertices w on the path connecting u and u_k which satisfy $1 \leq d(w, u_k) \leq 2^{k-1}\psi$.

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 PROC($P(u)$) is applied. The domains of the newly formed mobile packages resulting from the recursive application of Procedure 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 $Dom(P)$.

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

Case 6) If τ is of type add internal node, and u, the added vertex, belongs to $Dom(P)$, then u is added to domain $Dom(P)$ and the bottom most existing node in $Dom(P)$ 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 $Dom(P)$, then u is deleted but is considered to continues to belong to $Dom(P)$.

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 Procedure PROC($P(u)$). After Procedure PROC($P(u)$) is completed, for every $k \in \{0, 1, 2, \cdots, j(u) - 1\}$, 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 $k \in \{0, 1, 2, \cdots, j(u) - 1\}$. Let I_k denote that path containing the ancestors w of u satisfying $2^k\psi < d(u, w) \leq 2^{k+1}\psi$. The definition of $j(u)$ implies that before PROC($P(u)$) is applied, there is no mobile package of level k in path I_k, including any vertex in $Dom(P_k) \subset I_k$. Therefore, by the third domain invariant, $Dom(P_k)$ does not intersect with any other domain of a package of level k residing at a *descendant* of u_k. By the first and third domain invariants, and by the fact that $I_k \setminus Dom(P_k)$ does not contain any mobile package of level k either, we obtain that $Dom(P_k)$ 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. No change in a domain is needed in case 5 or when the topological event concerns nodes which are not in any domain and do not hold any package. hence, assume that τ is of type add internal node, and vertex u is added at time t between two existing vertices u and w. Case 6 above is applied. P's domain 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 $Dom(P)$ since the removed node is the bottom most. Similarly, the addition of the new node u keeps $Dom(P)$ as a path since u (and edges (v, u) and (u, w)) replace edge (v, w) on the path. Hence, the third invariant continues to hold.

If τ is of type remove leaf or remove internal node, and just before time t, the removed node u belonged to $Dom(P)$ for some package P, then u continues to belong to $Dom(P)$. 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, the third domain invariant holds too. This completes the proof. ∎

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,

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

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 it first moves its packages to its parent (see item 2 in the algorithm). 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 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 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 [1]. 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 [1], we defer the proof to the full paper.

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 is 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 full paper 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 (see [16, 7]) at u. If there is no static package at u then the agent climbs the tree (carried by messages) until it reaches a filler node or the root. It then takes the package located there and performs PROC(P) by walking down the tree.

The only real difference from the sequential setting is the fact that an agent cannot act instantaneously on multiple nodes, while in the 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 "graceful" 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 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. APPLICATIONS

In [1], the authors show how to use their (M, W)-controller to construct several basic schemes on growing trees (allowing only leaves to join). In particular, they show how to derive a size-estimation protocol and a *name-assignment* protocol (i.e., a protocol which assigns and maintains unique identities at the nodes of the changing tree, using $\log n + O(1)$ bits bet identity). Both their protocols (on growing trees) use $O(N\log^2 N)$ messages, where N is the maximum number of nodes in the tree.

It can be easily shown that the methods used in [1] for transforming their (M, W)-controller to protocols solving the above mentioned problems on growing trees, hold also for transforming our (M, W)-controller to protocols solving these problems in the more general dynamic model considered in this paper. Both resulted protocols use 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. Note, that in the restricted case where the tree can only grow, our protocols use the same message complexity as the corresponding protocols of [1].

We now demonstrate the use of our controller for other applications beyond the ones described in [1]. Specifically, we show how to use our size-estimation protocol for extending

various existing distributed data structures for *local queries* to dynamic settings.

We consider data structures in which, 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. Note, however, that constructing the data structure, or maintaining it when the network graph changes, requires communication. 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 our 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 [13]), 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 our 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 5.1. *The correctness of any ancestry labeling scheme (either dynamic or static) on trees is not affected by deletions of either internal nodes or leaves. (E.g., the dynamic ancestry schemes in [17, 20], and the static ones in [14].).*

CLAIM 5.2. *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 [13] and the flow and vertex connectivity labeling schemes in [15, 18, 19]).*
- *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. [17, 22]. 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 labels seems to be a more critical consideration than the total amount of storage needed for the information maintenance. In the following, when considering the memory complexity of a scheme, we only consider the size of labels $\mathcal{L}(v)$ and ignore the external memory $Memory(v)$ used for maintenance.

LEMMA 5.3. *Let π is one of the (either static or dynamic) data structures mentioned in either one of the above claims, and let $f(n)$ be an upper bound on the size of the labels (or routing tables) used in π. Let $\mathcal{M}(\pi, n)$ be an upper bound on message complexity used to assign the labels (or routing tables) of π on a* static *n-node network. Assume that $f(n)$ and $\mathcal{M}(\pi, n)$ are reasonable[1] functions. Then the following holds.*

1. *If π is one of the schemes mentioned in Claim 5.1, 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 $O(n_0 \log^2 n_0 + \mathcal{M}(\pi, n_0)) + O(\sum_i (\log^2 n_i + \frac{\mathcal{M}(\pi, n_i)}{n_i}))$ to the message complexity of π, per topological change.*
2. *If π be one of the schemes mentioned in Claim 5.2, 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 $O(n_0 \log^2 n_0 + \mathcal{M}(\pi, n_0) + \sum_i (\log^2 n_i + \frac{\mathcal{M}(\pi, n_i)}{n_i}))$ to the message complexity of π, per topological change.*

COROLLARY 5.4. *The ancestry schemes on trees mentioned in [17, 20, 14] can be extended to support also controlled deletions of both leaves and internal nodes, and the routing schemes on trees mentioned in [2, 12] can be extended to support also controlled deletions of leaves. The extended schemes have the same asymptotic label size of the original schemes, and their message complexity incurs an extra additive cost of $O(n_0 \log^2 n_0) + O(\sum_j \log^2 n_j)$ over the message complexity of the original schemes, where n_j is the number of nodes in the tree immediately after the j'th topological change.*

[1] A 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 $\alpha, \epsilon, \beta, \gamma > 0$

6. CONCLUSION

The memory complexity of the distributed controller 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 port numbers.

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 [1] 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 [1] uses messages of size $O(\log \log n)$. Intuitively, in our controller, an agent needs to count the number of steps it moves. Such counting was not necessary in the controller of [1] 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. Finally, we have shown that it is possible to match the message complexity of the controller of [1] 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.

7. 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**, (1996), 1–19.

[2] 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 (FOCS)*, 1989, 370–375.

[3] Y. Afek and M. E. Saks. Detecting Global Termination Conditions in the Face of Uncertainty. In *Proc. 6th Ann. ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing (PODC)* ,1987, pp. 109-124.

[4] B. Awerbuch, Y. Bartal, and A. Fiat. Competitive distributed file allocation. In *Proc. 25th Annual ACM Symposium on Theory of Computing (STOC)*, 1993, pp. 164-173.

[5] B. Awerbuch, S. Kutten, and D. Peleg. Competitive Distributed Job Scheduling. In *Proc. 24th Annual ACM Symposium on Theory of Computing (STOC)*, 1992, pp. 571-580.

[6] R. Bar-Yehuda and S. Kutten. Fault-Tolerant Majority Commitment. J. of Alg. Vol. 9, pp. 568–582, 1988.

[7] L. Barrire, P. Flocchini, P. Fraigniaud, and N. Santoro. Can we elect if we cannot compare? In *Proc. 15th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, 2003, pp. 324-332.

[8] Y. Bartal, A. Fiat, and Y. Rabani. Competitive Algorithms for Distributed data Management. J. Comput. Syst. Sci. 51(3): 341-358 (1995).

[9] Y. Bartal and A. Rosen. The Distributed k-Server Problem- A Competitive Distributed Translator for k-Server Algorithms. In *Proc. 33rd Symp. on Foundations of Computer Science (FOCS)*, pp. 344-354, 1992.

[10] E. W. Dijkstra. A note on two problems in connexion with graphs. Numerische Math. 1 (1959), S. pp. 269-271.

[11] 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).

[12] P. Fraigniaud and C. Gavoille. Routing in trees. In *Proc. 28th Int. Colloq. on Automata, Languages, and Programming (ICALP).* 2001, pp. 757–772.

[13] C. Gavoille, D. Peleg, S. Pérennes and R. Raz. Distance labeling in graphs. In *Proc. 12th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, 2001, pp. 210–219.

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

[15] M. Katz, N.A. Katz, A. Korman and D. Peleg. Labeling schemes for flow and connectivity. *SIAM Journal on Computing* **34** (2004),23–40.

[16] 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.

[17] A. Korman. General Compact Labeling Schemes for Dynamic Trees. In *Proc. 19th International Symposium on Distributed Computing (DISC)*, 2005.

[18] A. Korman. Labeling Schemes for Vertex Connectivity. In *Proc. 34th Int. Colloq. on Automata, Languages, and Programming (ICALP)*, 2007.

[19] A. Korman and S. Kutten. Distributed Verification of Minimum Spanning Trees. In *Proc. 25th Ann. ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing (PODC)*, 2006.

[20] A. Korman, D. Peleg, and Y. Rodeh. Labeling schemes for dynamic tree networks. Theory of Computing Systems (ToCS) 37 (2004), pp. 49-75.

[21] A. Korman and D. Peleg. Labeling Schemes for Weighted Dynamic Trees. In *Proc. 30th Int. Colloq. on Automata, Languages, and Programming (ICALP)*, 2003, pp. 369–383.

[22] A. Korman and D. Peleg. Dynamic Routing Schemes for General Graphs. In *Proc. 33rd Int. Colloq. on Automata, Languages, and Programming (ICALP)*, 2006.

[23] S. Kutten. Optimal Fault-Tolerant Distributed Construction of a Spanning Forest. Information Processing Letters, Vol. 27, pp. 299–307,May 1988.

[24] N.A. Lynch, N.D. Griffeth, M.J. Fischer and L.J. Guibas. Probabilistic Analysis of a Network Resource Allocation Algorithm. Inf. Cont. 68, 47-85.

[25] M.S. Manasse, L.A. McGeoch, and D.D. Sleator. Competitive Algorithms for On-Line Problems. In *Proc. 20th Annual ACM Symposium on Theory of Computing (STOC)*, 1988, pp. 322-333.

[26] J. Moy - 1994 - RFC 2328, April 1998.

[27] M. Thorup and U. Zwick. Compact routing schemes. In *Proc. 13th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, 2001, pp. 1-10.