A Local Algorithm for Ad Hoc Majority Voting via Charge Fusion

Liran Liss, Yitzhak Birk, Ran Wolff and Assaf Schuster
Technion — Israel Institute of Technology
E-mail: liranl@tx, birk@ee, ranw@cs, assaf@cs.technion.ac.il

***Abstract*— We present a local distributed algorithm for a general Majority Voting problem: different and time-variable voting powers and vote splits, arbitrary and dynamic interconnection topologies and link delays, and any fixed majority threshold. The algorithm combines a novel, efficient anytime spanning forest algorithm, which may also have applications elsewhere, with a "charge fusion" algorithm that roots trees at nodes with excess "charge" (derived from a node's voting power and vote split), and subsequently transfers charges along tree links to oppositely charged roots for fusion. At any instant, every node has an ad hoc belief regarding the outcome. Once all changes have ceased, the correct majority decision is reached by all nodes, within a time that in many cases is independent of the graph size. The algorithm's correctness and salient properties are proved, and experiments with up to a million nodes provide further validation and actual numbers. To our knowledge, this is the first locality-sensitive solution to the Majority Vote problem for arbitrary, dynamically changing communication graphs.**

I. Introduction

A. Background

Emerging large-scale distributed systems, such as the Internet-based peer-to-peer systems, grid systems, ad hoc networks, and sensor networks, impose uncompromising scalability requirements on (distributed) algorithms used for performing various functions. Clearly, for an algorithm to be perfectly scalable, i.e., O(1) complexity in problem size, it must be "local" in the sense that a node only exchanges information with nodes in its vicinity. Also, information must not need to flow across the graph. For some problems, there are local algorithms whose execution time is independent of the graph size. Examples include Ring Coloring [1] and Maximal Independent Set [2].

Unfortunately, there are important problems for which there cannot be such perfectly-scalable solutions. Yet, locality is a highly desirable characteristic: locality decouples computation from the system size, thus enhancing scalability; also, handling the effects of input changes or failures of individual nodes locally cuts down resource usage and prevents hot spots; lastly, a node is usually able to communicate reliably and economically with nearby nodes, whereas communication with distant nodes, let alone global communication, is often costly and prone to failures.

With these motivations in mind, efficient local (or "locality sensitive") algorithms have also been developed for problems that do not lend themselves to solutions whose complexity is completely independent of the problem instance. One example is an efficient Minimum Spanning Tree algorithm [2]. Another example is fault-local mending algorithms [3], [4]. There, a problem is considered fault-locally mendable if the time it takes to mend a batch of transient faults depends only on the number of failed nodes, regardless of the size of the network. However, the time may still be proportional to the size of the network for a large number of faults.

The notion of locality that was proposed in [3], [4] for mending algorithms can be generalized as follows: an algorithm is local if its execution time does not depend on the system size, but rather on some other measure of the problem instance. The existence of such a measure for non-trivial instances of a problem suggests (but may not guarantee) the possibility of a solution with unbounded scalability (in graph size) for these instances. This observation encourages the search for local algorithms even for problem classes that are clearly global for some instances. In this paper, we apply this idea to the Majority Vote problem, which is a fundamental primitive in distributed algorithms for many common functions; E.g., leader election, consensus and synchronization.

B. The Majority Vote Problem

Consider a system comprising an unbounded number of nodes, organized in a communication graph. Each node has a certain (possibly different) voting power on a proposed resolution, and may split its votes arbitrarily between "Yes" and "No". Nodes may change their connectivity (topology changes) at any moment, and both the voting power and the votes themselves may change over time[1]. In this dynamic setting, we want every node to decide whether the fraction of Yes votes is greater than a given threshold. Since the outcome is inherently ad hoc, it makes no sense to require that a node be aware of its having learned the "final" outcome, and we indeed do not impose this requirement. However, we do require eventual convergence in each connected component.

The time to determine the correct majority decision in a distributed vote may depend on the significance of the majority rather than the system size. In certain cases such as a tie, computing the majority would require collecting at least half of the the votes, which would indeed take time proportional to the size of the system. Yet, it appears possible that whenever the majority is evident throughout the graph, computation can be extremely fast by determining the correct majority decision based on local information alone.

Constantly adapting to the input in a local manner can also lead to efficient anytime algorithms: when the global majority changes slowly, every node can track the majority decision in a timely manner, without spending vast network resources; when a landslide majority decision flips abruptly due to an instant change in the majority of the votes, most of the nodes should be able to reach the new decision extremely fast as discussed above; and, after the algorithm has converged, it should be possible to react to a subsequent vote change that increases the majority with very little, local activity. A less obvious situation occurs when a vote change reduces the majority (but does not alter the outcome), because the change may create a local false perception that the outcome has changed as well. The challenge to the algorithm is to squelch the wave of erroneous perceived outcome fast, limiting both the number of affected nodes and the duration of this effect.

The Majority Vote problem thus has instances that require global communication, instances that appear to lend themselves trivially to efficient, local solutions, and challenging instances that lie in between.

The main contribution of this paper is a local algorithm for the Majority Vote problem. Our algorithm comprises two collaborating components: an efficient anytime spanning forest algorithm and a charge-fusion mechanism. A node's initial charge is derived from its voting power and vote split such that the majority decision is determined by the sign of the net charge in the system. Every node bases its ad-hoc belief of the majority decision according to the sign of its charge or that of a charged node in its vicinity. The algorithm roots trees at charged nodes, and subsequently fuses opposite charges using these trees until only charges of one (the majority) sign are left, thus disseminating the correct decision to all nodes.

[1]Nodes are assumed to trust one another. We do not address Byzantine faults in this paper.

We provide formal proofs for key properties as well as simulation results that demonstrate actual performance and scalability. Offering a preview of our results, our experiments show that for a wide range of input instances, the majority decision can be computed "from scratch" in constant time. Even for a tight vote of 52% vs. 48%, each node usually communicates with only tens of nearby nodes, regardless of the system size. In [5], similar behavior was demonstrated using an (unrelated) algorithm that was suited only for tree topologies. To our knowledge, the current paper offers, for the first time, a locality-sensitive solution to the Majority Vote problem for arbitrary, dynamically changing communication graphs.

The remainder of the paper is organized as follows: In Section II we provide an overview of our approach. Section III presents our spanning forest (SF) algorithm, and our majority vote (MV) algorithm is detailed in section IV. In section V, we provide some empirical results to confirm our assumptions and demonstrate the performance of our algorithm. Section VI describes at some length previous related work. We conclude the paper in Section VII. Due to space limitation, full proofs are deferred to the Appendix.

II. Overview of Our Approach

Consider a vote on a proposition. The voting takes place at a set of *polls*, which are interconnected by communication links. We propose the following simple protocol for determination of the global majority decision. For each unbalanced poll, transfer its excess votes to a nearby poll with an opposite majority, leaving the former one balanced. Every balanced poll bases its current belief regarding the majority decision on some unbalanced poll in its vicinity. We continue this poll consolidation process until all remaining unbalanced polls posses excess votes of the same type, thus determining the global majority decision. We next state the problem formally, and elaborate on our implementation of the foregoing approach.

Let *G(V,E)* be a graph, and let $\lambda = \lambda_n/\lambda_d$ be a rational threshold between 0 and 1. Every node *i* is entitled to

V_i votes; we denote the number of node *i*'s Yes votes by Y_i. For each connected component *X* in *G*, the desired majority vote decision is Yes if and only if the fraction of Yes votes in *X* is greater than the threshold:

$$\frac{\sum_{i\in X} Y_i}{\sum_{i\in X} V_i} > \lambda.$$

Since a node can change its current vote in any time, we need to distinguish between a node's current vote and the votes or "tokens" that we transfer between nodes during the consolidation process. In order to prevent confusion, we introduce the notion of the *("electrical") charge* of a node, and base the majority decision on the sign of the net charge in the system. The following equivalent criterion for determining a Yes majority vote decision allows us to work with integers and only deal with linear operations (addition and subtraction):

$$\lambda_d \sum_{i\in X} Y_i - \lambda_n \sum_{i\in X} V_i > 0.$$

A node *i*'s charge, C_i, is initially set to $\lambda_d Y_i - \lambda_n V_i$. Subsequent single-vote changes at a node from No to Yes (Yes to No) increase (decrease) its charge by λ_d. An addition of one vote to the voting power of a node reduces its charge by λ_n if the new vote is No, and increases it by $\lambda_d - \lambda_n$ if the vote is Yes. A reduction in a node's voting power has an opposite effect. Charge may also be transferred among nodes, affecting their charges accordingly but leaving the total charge in the system unchanged. Therefore, the desired majority vote decision is Yes if and only if the net charge in the system is non-negative:

$$\sum_{i\in X} C_i \geq 0.$$

Our Majority Vote algorithm (MV) entails transferring charge among neighboring nodes, so as to "fuse" and thereby eliminate equal amounts of opposite-sign charges and, in so doing, also relay ad hoc majority decision information. Eventually, all remaining charged nodes have an identical sign, which is the correct global majority decision. Therefore, if we could transfer charge such that *nearby* charged nodes with opposite signs canceled one another without introducing a livelock, and subsequently disseminate the resulting majority decision to neutral nodes locally, we would have a *local* algorithm for the Majority Vote problem.

We solve the aforementioned livelock problem with the aid of a local spanning forest algorithm (SF) that we will introduce shortly. The interplay between SF and MV is as follows. The roots of SF's trees are set by MV at charged nodes. SF gradually constructs distinct trees over neutral nodes. MV then deterministically routes charges of one sign over directed edges of the forest constructed by SF towards roots containing opposite charge. The charges are fused, leaving only their combined net charge. Finally, MV unroots nodes that turned neutral, so SF will guarantee that all neutral nodes join trees rooted at remaining charged ones in their vicinity. Each node bases its (perceived global) majority decision on the charge sign of its tree's root. Therefore, disseminating a majority decision to all nodes is inherently built into the algorithm.

We note that although the system is dynamic, we ensure that the total charge in any connected component of the graph always reflects the voting power and votes of its nodes. By so doing, we guarantee that the correct majority decision is eventually reached by every node in any given connected component, within finite time following the cessation of all changes.

III. Spanning Forest Algorithm

In this section, we describe SF, an efficient algorithm for maintaining a spanning forest in dynamic graphs, and prove its loop-freedom and convergence properties. In the next section, we will adapt this algorithm and utilize it as part of MV.

A. SF Algorithm description

Given a (positive) weighted graph and a set of nodes marked as *active* roots, the algorithm gradually builds trees from these nodes. At any instant, edges and nodes can be added or removed, edge weights can change, and nodes can be marked/unmarked as active roots on the fly. However, the graph is always loop-free and partitioned into distinct trees. Some of these trees have active roots, while others are either inactive singletons (the initial state of every node) or rooted at nodes that used to be active. We denote a tree as *active* or *inactive* based on the activity state of its root.

Whenever the system is stable, each connected component converges to a forest in which every tree is active (if active roots exist). Loop freedom ensures that any node whose path to its root was cut off, or whose root became inactive, will be able to join an active tree in time proportional to the size of its previous tree. Unlike shortest path routing algorithms that create a single permanent tree that spans the entire graph (for each destination), SF is intended to create multiple trees that are data-dependent, short-lived, and local. Therefore, in order to reduce control traffic, an edge-weight change does not by itself trigger any action. Nevertheless, expanding trees do take into account the most recent edge weight information. So, although we do not always build a shortest path forest, our paths are short.

Algorithm 1 (*see end of paper*) presents SF. In addition to topological changes, the algorithm supports two operations to specify whether a node should be treated as an active root ($Root_i$ and $UnRoot_i$), and one query ($NextHop_i$) that returns a node's downtree neighbor, or $\perp$ if the node is a root. (We denote by downtree the direction from a node towards its root.) To its neighbors, a node i's state is represented by its perceived tree's activity state T_i, its current shortest path weight W_i, and an acknowledgement number A_i. The algorithm converges in a similar manner to Bellman-Ford algorithms [6]: after each event, node i considers changing its next hop pointer (P_i) to a neighbor that minimizes the weight of its path to an active root (step 2b). More formally, to a neighbor j that is believed by i to be active ($\lambda_i(T_j) = 1$) and for which $\lambda_i(W_j) + d(i, j)$ is minimal.

Loops are prevented by ensuring that whenever a portion of a tree is inactivated due to an $UnRoot$ operation or a link failure, a node will not point to a (still active) node that is uptree from it [7]. (Edge weight increases can also cause loops. However, we do not face this problem because such increases do not affect a node's current weight in our algorithm.) This is achieved both by limiting a node i's choice of its downtree node (next hop) to neighbors that reduce i's current weight, and by allowing i to increase its current weight only when i and all its uptree nodes are inactive (step 2a).

In order to relay such inactivity information, we use an acknowledgement mechanism as follows: a node i will not acknowledge the fact that the tree state of its downtree neighbor has become inactive (step 4), before i is itself inactivated (T_i is set to 0 and A_i is incremented in step 2c) and receives acknowledgements for its own inactivation from all its neighbors ($IsAck(i)$ becomes $true$). Note that i will acknowledge immediately an inactivation of a neighbor that is not its downtree node. Therefore, if a node i is inactive and has received the last corresponding acknowledgement, all of i's uptree nodes must be inactive and their own neighbors are aware of this fact.

An active root expands and shrinks its tree at the fastest possible speed according to minimum path considerations. However, once a root is marked inactive, it takes a three-phase process to mark all nodes in the corresponding tree as inactive and reset their weight to ∞. First, the fact that the tree is inactive ($T_i = 0$) propagates to all the leaves. Next, Acks are aggregated from the leaves and returned to the root. Note that node weights remain unchanged. Finally, the root increases its weight to ∞. This weight increase propagates towards the leaves, resetting the weight of all nodes in the tree to ∞ on its way. It may seem that increasing the weight of the leaves only in the third phase is wasteful. However, this extra phase actually speeds up the process by ensuring that nodes in "shorter" branches do not choose as their next hop nodes in "longer" branches, which haven't yet been notified that the tree is being inactivated. (This phase corresponds to the $wait$ state in [7].)

B. Loop Freedom

For facility of exposition, given a node i we define $\widehat{W_i}$ to equal W_i if $T_i = 1$, and ∞ otherwise.

Lemma 1: After any event in which $\widehat{W_i}$ increases for some nonisolated node i, $\widehat{W_i} = \infty$, $T_i = 0$ and $IsAck(i) = false$.

Proof: Follows directly from the algorithm. ■

Lemma 2: If $P_i \neq \perp$, either $\widehat{W_i} > \lambda_i(\widehat{W_{P_i}})$ or $\widehat{W_i} = \lambda_i(\widehat{W_{P_i}}) = \infty$.

Proof: By induction on events. ■

Lemma 3:
1) For every node i, if $IsAck(i) = true$ then for every j uptree from i or $j = i$, and for every neighbor m of j: (a) $\lambda_m(\widehat{W_j}) \geq \widehat{W_i}$, and (b) for every in-transit update message u sent by j with weight $\widehat{W_u}$: $\widehat{W_u} \geq \widehat{W_i}$.
2) For every node i, if $IsAck(i) = false$ then the same claims hold when replacing $\widehat{W_i}$ with W_i.

Proof: By induction on events. ■

Theorem 1: There are no cycles in the graph at any instance.

Proof: Let i be a node that closes a cycle at time t_0. Therefore, at t_o^+ we have $\lambda_i(\widehat{W_{P_i}}) = \lambda_i(W_{P_i}) < W_i = \widehat{W_i}$. According to 1) or 2) of Lemma 3, $\lambda_i(\widehat{W_{P_i}}) \geq \widehat{W_i}$, since P_i is also uptree from i. A contradiction. ■

C. Convergence

We assume that the algorithm was converged at time 0, after which a finite number of topological and root changes occurred. Let t_0 be the time of the last change.

Lemma 4: There exists a time $t_1 > t_0$ s.t. for every $t > t_1$, if $IsAck(j)$ changes from $true$ to $false$ for some node j, there exists a node i for which $IsAck(i) = false$ and $W_i < W_j$.

Lemma 5: If $IsAck(i) = false$ for some node i, $IsAck(i)$ will change to $true$ in finite time.

Proof: By contradiction. ■

Lemma 6: There exists a time $t_2 > t_0$, s.t. for every $t > t_2$ $IsAck = true$ for all nodes.

Theorem 2: The algorithm converges in finite time after all topological and root changes have stopped.

Proof: Let t_2 be the time stated in Lemma 6. After this time, there are no restrictions on choosing next hops. The algorithm can then be reduced to a normal Bellman-Ford algorithm, in which remaining active roots are simulated by zero weighted edges connected to a single destination node. Therefore, normal proofs for Bellman-Ford algorithms apply here [8]. ■

IV. Majority Vote

In this section, we first describe the required adaptations to SF for use in our Majority Vote algorithm (MV). Then we provide a detailed description of MV, followed by a correctness proof. Finally, we prove MV's locality properties.

A. SF adaptation

We augment SF as follows:

1) To enable each neutral node to determine its majority decision according to its tree's root, we expand the SF root and tree state binary variables (R_i and T_i) to include the value of -1 as well. While inactive nodes will still bear the value of $T = 0$, the tree state of an active node i will always equal the sign of its next hop (downtree neighbor) as known to i: $T_i = \lambda_i(T_{P_i})$ or the sign of R_i if i itself is an active root.
2) We attach a "Tree ID" variable to each node for symmetry breaking as explained next. It is assigned a value in every active root, and this value is propagated throughout its tree.
3) To enable controlled routing of charge from the root of one tree to that of an opposite-sign tree that collided with it, each node also maintains an *inverse hop,* which designates a weighted path to the other tree's root.
   Node i considers a neighbor j as a candidate for its inverse hop in two cases: (a) i and j belong to different trees and have opposite signs ($T_i = -T_j$); (b) i is j's next hop, both nodes have the same sign ($T_i = T_j$), and j has an inverse hop. We further restrict i's candidates only to those designating a path towards a root with a higher Tree ID. (Different IDs ensure that only one of the colliding trees will develop inverse hops.) If there are remaining candidates, i selects one that offers a path with minimal weight, or $\perp$ otherwise.
4) To guarantee that paths do not break while routing charges, we prevent an active node from changing its next hop. However, as will be explained shortly, there are cases where new active roots should be able to take over nodes of neighboring active trees. Therefore, we extend the *Root* operation to include an *expansion flag*. Setting this flag creates a one-shot expansion wave, by repeatedly allowing any neighboring nodes to join the tree. The wave will die out when it stops improving the shortest path of neighboring nodes.

TABLE I
SF Interface

Procedure	Function
$Root_i\,(sign, ID, expand)$	Mark i as an active root with a corresponding sign, ID, and expansion property
$UnRoot_i$	Unmark i as an active root
$TreeSign_i$	Return i's tree state
$TreeID_i$	Return i's tree ID
$NextHop_i$	Return i's next hop, or $\perp$ if i is a root
$InvHop_i$	Returns i's preferred inverse hop, or $\perp$ if there is none

Proposition 1: The adaptations above do not invalidate the correctness or the convergence of the SF algorithm.

Proof: 1-3) merely add information, and hence do not affect the behavior of the algorithm. 4) only restricts next-hop choices of active nodes. Therefore, 4) cannot cause loops. Finally, inactive nodes can always join active trees. So when SF stops, all nodes will belong to active trees as required. ■

The interface of the augmented SF algorithm exposed to MV is summarized in Table I.

B. MV Algorithm description

MV is an asynchronous reactive algorithm. It operates by expressing local vote changes as charge, relaying charge sign information among neighboring nodes using SF, and fusing opposite charges to determine the majority decision based on this information. Therefore, both events that directly affect the current charge of a node, and events that relay information on neighboring charges (via SF), cause an algorithm action.

Every distinct charge in the system is assigned an ID. The ID need not be unique, but positive and negative charges must have different IDs (e.g., by using the sign of a charge as the least significant bit of its ID). Whenever a node remains charged following an event, it will be marked as an active root (using the SF $Root$ operation), with the corresponding sign and charge ID. If the event was a vote change, we also set the root's expansion flag to balance between the size of the new tree and its neighborhood. This improves overall tree locality, since a vote change has the potential of introducing a new distinct charge into the system.

When trees of opposite signs collide, one of them (the one with the lower ID) will develop inverse hops as explained above. Note that inverse hops are not created arbitrarily: they expand along a path leading directly to the root. Without loss of generality, assume that the negative tree develops inverse hops. Once the negative root identifies an inverse hop, it sends all its charge (along with its ID) to its inverse hop neighbor and subsequently unmarks itself as an active root (using the SF $UnRoot$ operation). The algorithm will attempt to pass the charge along inverse hops of (still active) neutral nodes that belonged to the negative tree (using the SF $InvHop$ query), and then along next hops of nodes that are part of the positive tree (using the SF $NextHop$ query).

As long as the charge is in transit, it does not develop a new root. If it reaches the positive root, fusion takes place. The algorithm will either inactivate the root or update the root's sign and charge ID, according to the residual charge. In case the propagation was interrupted (due to topological changes, vote changes, expanding trees, etc.), the charge will be added to that of its current node, possibly creating a new active root.

Algorithm 2 (*see end of paper*) details MV formally. $C_i(j)$ keeps track of every charge transferred between a node and each of its neighbors. It is used to ensure that charges remain within the connected component they were generated. $GenID(charge)$ can be any function that returns a positive integer, as long as different IDs are generated for positive and a negative charges. However, we have found it beneficial to give higher IDs to charges with greater absolute values, which will cause them to ”sit in place” as roots. This scheme results in faster fusion since charges with opposite signs and lower absolute values will be routed towards larger charges in parallel. It also discourages fusion of large identical-sign charges when multiple charges in transit overwhelm a common destination node, before the algorithm propagates its new state.

After updating a node i's charge information following an event, the algorithm performs two simple steps. In step 1, if i is charged, the algorithm attempts to transfer the charge according to i's tree sign and current next/inverse hop information obtained from SF. In step 2, i's root state is adjusted according to its remaining charge. The output of the algorithm, i.e., the estimated majority decision at every node, is simply the sign of the node's tree state (using SF's $TreeSign$ query). For inactive nodes, we arbitrarily return $true$.

C. Correctness

We prove the correctness and termination of the algorithm when the system is stable using the following simple lemmas. Assume that all external events (link state changes, bit changes, etc.) stop at some time t_0.

Lemma 7: There exists a time $t_1 > t_0$ for which a node can change its next hop only if its corresponding tree root is inactive.

Lemma 8: For every $t > t_1$, the charge of a neutralized node (a node that has been robbed of its charge) will reach a neighboring charged root with an opposite sign if this root itself is not neutralized as well during this time.

Lemma 9: Algorithm 2 stops in finite time after t_1

Lemma 10: When the algorithm stops, either all charged nodes are positive, or all charged nodes are negative.

Lemma 11: For any connected component *X*, if no transfer messages are underway then:

$$\sum_{i \in X} C_i = \lambda_d \sum_{i \in X} Y_i - \lambda_n \sum_{i \in X} V_i$$

Theorem 3: Algorithm 2 stops in finite time after all external events have ceased with the correct output in every node.

Proof: Termination is guaranteed from Lemma 9. Let X be a connected component after the algorithm stopped. Assume that the majority decision for all nodes in X should be $true$, i.e., $\lambda_d \sum_{i \in X} Y_i - \lambda_n \sum_{i \in X} V_i \geq 0$. Hence, according to Lemma 11, $\sum_{i \in X} C_i \geq 0$. It follows from Lemma 10 that $\forall i \in X : C_i \geq 0$. Since there are no negative trees, all nodes in X decide $true$. The corresponding case is shown similarly. ■

D. Locality properties

The locality of an execution of the algorithm depends on the input instance. In all cases in which the majority is evident throughout the graph, the algorithm takes advantage of this by locally fusing minority and majority charges in parallel. Many input instances follow this pattern, especially when the majority is significant.

The algorithm operates in a way that preserves the charge distribution because: 1) further vote changes create new roots uniformly, and 2) our charge ID scheme discourages fusion of charges of the same sign. Therefore, we conjecture that for many input instances, the size of remaining trees after the algorithm has converged will be determined by the majority percentile, rather than by the graph size. For example, consider a fully connected graph of size N for which each node has a single vote, a threshold of 1/2, and a tight vote of 48% vs. 52%. After the algorithm converges, the absolute net charge is $4\% \cdot N$. Assuming that the remaining charge is spread uniformly so that every charge unit establishes an active root of its own, the number of nodes in each tree is about $\frac{N}{4\% \cdot N} = 25$, regardless of whether the graph contains a hundred or a million nodes.

From this conjecture it follows that, for these instances, there exists a non-trivial upper bound R on the radius of any tree in the graph. We initially prove that the algorithm is local for single vote changes and when several changes occur far from one another. We then show that the algorithm is local for any fixed number of changes. In the next section, we will use simulations to verify our conjecture empirically, and demonstrate the local characteristics of our algorithm for arbitrary vote changes.

Lemma 12: Let E be an environment for which the algorithm has converged with sign S (all trees are of the same sign), and let R be an upper bound on the radius of any tree in E. If a single node i in E forms a new tree of opposite sign and maximal ID then, after at most R time steps, the growth rate of the new tree's radius is at most half the one-way propagation speed.

Theorem 4: Assume that all vote and topological changes have stopped, and MV has converged. Let R be an upper bound on the radius of any tree in the graph. If some node changes a single vote, then the algorithm convergence time is a function of R, independent of the overall graph size.

Corollary 1: Assume that all vote and topological changes have stopped, and MV has converged. Let R be an upper bound on the radius of any tree in the graph. If vote changes occur at multiple nodes such that the resulting protocol actions do not coincide with one another, then the algorithm convergence time is a function of R.

For the general case of a number of vote changes, we do not give a bound on convergence time. However, we show that the algorithm is still local by proving finite convergence time even for infinite graphs.

Theorem 5: Let G be an infinite graph, and let R be an upper bound on the radius of any tree in the graph. Also, assume that MV has converged following the cessation of all changes. If $m < \infty$ vote changes occur and do not change the majority decision then the algorithm converges in finite time.

V. Empirical study

We simulated the algorithm's execution on large graphs. The coded algorithm includes several details, such as Ack management, that were partly omitted from the discussion for brevity, as well as various small local optimizations that do not alter correctness. We examined the time required until various levels of convergence are achieved (in terms of the fraction of nodes that have reached the correct outcome and do not retract), as well as the mean number of messages per edge. For simplicity, we only considered a 50% majority threshold and one vote per node. However, simulations were run for several Yes/No voting percentages, thereby checking the sensitivity of the results to the proximity of the vote to the decision threshold.

Two representative graph topologies were used: a mesh for computing centers and sensor networks, and de Bruijn graphs for structured peer-to-peer systems [9]. For each, graph sizes varied from 256 nodes to 1024K nodes. Finally, both "bulk" ("from scratch") voting and ongoing voting were simulated.

In bulk mode, all nodes voted simultaneously at $t = 0$ with the desired percentage of Yes votes, and we measured the time until various fractions (90%, 95%, 100%, etc.) of the nodes decided on the correct outcome without subsequently retracting. Multiple experiments were carried out for a (graph type, size, Yes fraction) combination, with i.i.d drawings of the votes in the different experiments, and the results were averaged.

Figure 1 (a) and (b) depict the results for a percentile of 95% and 100% for graphs with 256 to 1024K nodes and several Yes/No ratios. As can be seen from Figure 1(a), the time it takes for 95% of the nodes to reach the correct outcome depends only on the percentage of Yes votes and is independent of graph size. This is evidence of the algorithm's local behavior. Figure 1(b) presents the time for 100% convergence, i.e., the time until the last node reaches the correct outcome. This measurement is deemed to be very noisy. When averaging the results over several runs, we observe that for de Bruijn graphs, the time to 100% convergence is nearly constant regardless of graph size. For mesh graphs, the time appears proportional to the logarithm of graph size. Note that

Fig. 1. Bulk mode convergence and scale-up

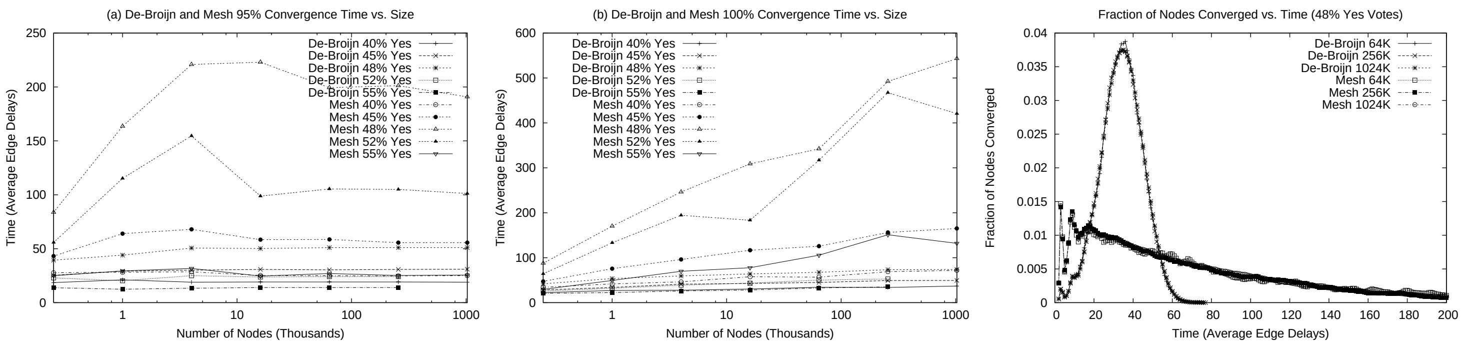
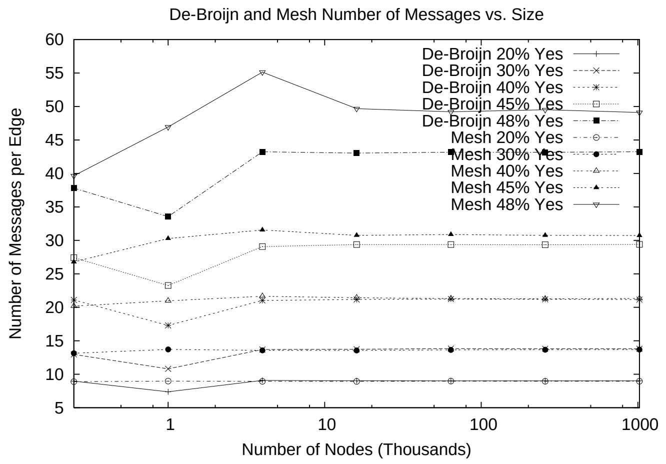


Fig. 2. Messages per edge



this worst case (over graph nodes) result is nonetheless averaged over multiple voting instances.

Figure 1 (c) focuses on the "convergence percentile", providing the probability density of "converged" nodes over time. Two things are readily evident from the figure: 1) beyond the mean time to convergence, the number of unconverged nodes declines exponentially with time; 2) this distribution is independent of graph size. In fact, the distributions for different graph sizes barely distinguishable. This strongly suggests taht locality and scalability hold for virtually every percentile except 100%.

Next, we investigated the communication resources consumed by the algorithm. We measured the number of messages per edge versus graph size and the fraction of Yes voters. As depicted in Figure 2, the number of messages per edge depends only on the percentage of Yes votes and not on graph size.

Our algorithm may send up to 50 messages per edge, compared with only two in the optimal centralized algorithm. However, in our algorithm those messages are sent concurrently throughout the graph and nodes do not wait for one another. In many practical scenarios (e.g., token ring protocols), refraining from sending a message is itself wasteful.

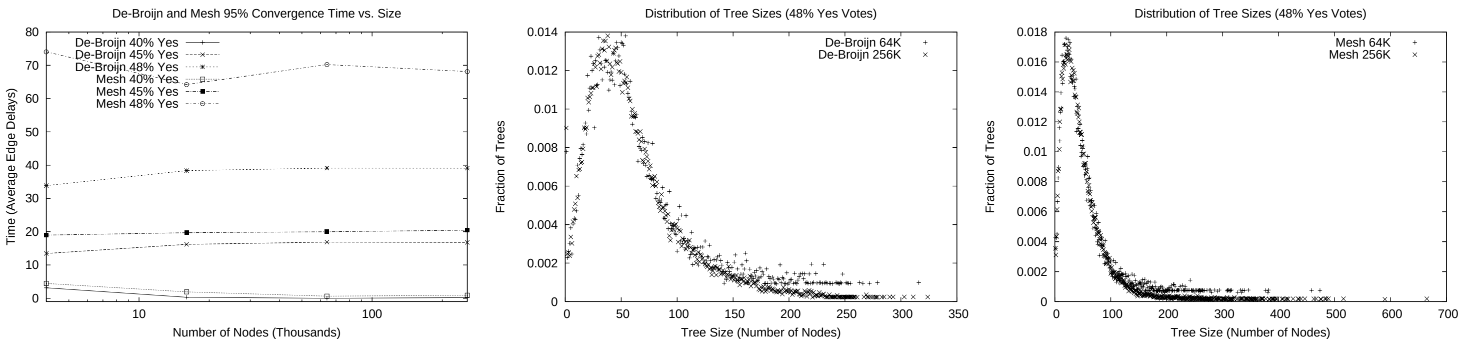
So far, we only considered bulk voting. In our last set of experiments we investigated an ongoing operation. Here, a given fraction (0.1%) of the nodes changes its vote with every message that is sent. However, the overall percentage of Yes votes remains the same. We view this operation mode as the closest to real-life. In this setting we wish to evaluate the time it takes for the effect of a single change to subside and to validate that our algorithm does not converge to some pathological situation. An example of a pathological situation is one in which all charge converges at a single node, whose tree then spans the entire graph.

In these experiments, we ran the system for some time. Subsequently, we stopped all changes and made two measurements: the time it takes for the system to converge, and the number of nodes in each tree upon convergence. As expected, convergence time (Figure 3(a)) in on-going mode does not differ from convergence in bulk mode (Figure 1(a)). As depicted in Figures 3(b)(c), tree sizes are tightly distributed about their mean. There are only few large trees, the largest of which spans approximately one percent of the graph. These experiments thus confirm our conjecture that tree sizes are small, and demonstrate that locality is maintained in the on-going mode as well.

VI. Related Work

Our work bears some resemblance to Directed Diffusion [10], a technique to collect aggregate information

Fig. 3. On-going mode convergence and locality



in sensor networks. As in their work, our routing is data-centric and based on local decisions. However, our induced routing tables are relatively short-lived, and do not require refreshments or enforcements. The SF algorithm we present, builds upon previous research in distributed Bellman-Ford routing algorithms which avoid loops such as [7] and [8].

Several alternative approaches can be used to conduct majority voting such as sampling, pseudo-static computation, and flooding. With sampling the idea is to collect data from a small number of nodes selected with uniform probability from the system, and compute the majority based on that sample. One such algorithm is the gossip based work of Kempe et al. [11]. Yet sampling can not guarantee correctness and is sensitive to biased input distributions. Moreover, gossip based algorithms make assumptions on the mixing properties of the graph which do not hold for any graph. Pseudo-static computation suggests to perform a straightforward algorithm that would have computed the correct result had the system been static, and then bound the error due to possible changes. Such is the work by Bawa et. al. [12] for example. In flooding, input changes of each node are flooded over the whole graph, so every node can compute the majority decision directly. Simple as it may sound, flooding guarantees convergence to an exact solution in stable periods. However, the communication costs of flooding are immense. Furthermore, the memory requirements of the method are proportional to the size of the system.

One related problem which has been addressed by local algorithms is the problem of local mending or persistent bit. In this problem all nodes have a state bit which is initially the same. A fault changes a minority of the bits and the task of the algorithm is to restore the bits to their initial value. A local solution was given for this problem in [3], which is correct so long as the size of the minority is smaller than $N/\log N$. Our algorithm can solve the same problem for any size of the minority. Another algorithm for this problem was given in [4]. This second algorithm accepts a minority of any size. However, it only works for a static topology and with lockstep execution. Our algorithm, in contrast, allows topology changes and asynchronous communication.

Finally, [5] also conducts majority votes in dynamic settings. However, their algorithm assumes the underlying topology is a spanning tree. Although this algorithm can be layered on top of another distributed algorithm that provides a tree abstraction, a tree overlay does not make use of all available links as we do, and its costs must be taken into account. Even when assuming that once a tree is constructed its links do not break, simulations have shown that while [5] is faster in cases of a large majority, our algorithm is much faster as the majority is closer to the threshold.

VII. Conclusions

We presented a local Majority Vote algorithm intended for dynamic, large-scale asynchronous systems. It uses an efficient, anytime spanning forest algorithm as a subroutine, which may also have other applications. The Majority Vote algorithm closely tracks the ad hoc solution, and rapidly converges to the correct solution upon cessation of changes. Detailed analysis revealed that if the occurrences of voting changes are randomly and uniformly spread across the system, the performance of the algorithm depends only on the number of changed votes and the current majority size, rather than the system size. A thorough empirical study demonstrated the excellent scalability of the algorithm for up to millions of nodes – the kind of scalability that is required by contemporary distributed systems.

APPENDIX

A. Loop freedom of the Spanning Forest algorithm (SF)

Proof of Lemma 1

Follows directly from the algorithm. Increases in W_i are possible only when $T_i = 0$ (step 2a), and therefore do not affect $\widehat{W_i}$ whose value is already ∞. Therefore, the only increase in $\widehat{W_i}$ is due to T_i becoming 0, setting $\widehat{W_i}$ to ∞. Because i is nonisolated and A_i is incremented, $IsAck(i) = false$ (step 2c).

Proof of Lemma 2

By induction on events. Initially all nodes are isolated, so the Lemma holds trivially. Consider an event at node i at $t = t_0$. We will show that if the Lemma holds at t_0^-, it will also hold at t_0^+. If i changes its next hop at t_0 to some node j or $\lambda_i(W_j)$ has not increased for an existing next hop j, step 2b ensures that $W_i = \lambda_i(W_j) + d(i, j)$. Therefore, $\lambda_i(\widehat{W_{P_i}}) < \widehat{W_i} < \infty$ because $T_i = \lambda_i(T_j) = 1$ and $d(i, j) > 0$. Otherwise, $\lambda_i(T_j)$ has increased for an existing next hop j. Following Lemma 1, $\lambda_i(\widehat{W_j})$ must be ∞, causing $\widehat{W_i}$ to also be set to ∞ in step 2c.

Proof of Lemma 3

By induction on events. Initially all nodes are isolated so the Lemma holds trivially. Consider an event at t_0, and assume that the Lemma was correct with respect to some node i at t_0^-. We show that the Lemma still holds at t_0^+ by contradiction. The Lemma can be violated in the following cases:

1) The event occurs in i. Since for any update message u sent by i at t_0^+: $\widehat{W_u} = \widehat{W_i}(t_0^+)$, the Lemma can be violated only due to update messages in transit from i or due to nodes uptree from i, as a result of i's new state. We reach a contradiction by examining all $IsAck(i)$'s possible state transitions:
   - $IsAck(i)(t_0^-) = IsAck(i)(t_0^+) = true$. Lemma 1 guarantees that $\widehat{W_i}$ cannot increase. Therefore, 1) cannot be violated.
   - $IsAck(i)(t_0^-) = IsAck(i)(t_0^+) = false$. Since W_i can only increase in step 2a, and $IsAck(i)(t_0^-) = false$, W_i cannot increase. Therefore, 2) cannot be violated.
   - $IsAck(i)$ changes from $true$ to $false$. Since this change can only occur if $T_i(t_0^-) = 1$, it follows from the algorithm that W_i cannot change, and we have: $\widehat{W_i}(t_0^-) = W_i(t_0^-) = W_i(t_0^+)$. Therefore, 2) holds at t_0^+ due to our assumption that 1) held at t_0^-.
   - $IsAck(i)$ changes from $false$ to $true$. This change can only occur if the event is the reception of an Ack message for i's latest inactivation. If $T_i(t_0^-) = 1$, it follows from the algorithm that W_i cannot change, and we have: $W_i(t_0^-) = W_i(t_0^+) = \widehat{W_i}(t_0^+)$. Therefore, 1) holds at t_0^+ due to our assumption that 2) held at t_0^-. If $T_i(t_0^-) = 0$, every node j uptree from i has set $T_j = 0$, and received an acknowledgement from all its neighbors before returning an acknowledgement to its downtree node. Therefore, for every j uptree from i at t_0, $\widehat{W_j} = \infty$ and there are no update messages in transit from j. Likewise, for every neighbor m of j at t_0: $\lambda_m(\widehat{W_j}) = \infty$. 1) holds trivially.
2) The Lemma is violated by a change in $\lambda_m(\widehat{W_j})$, where m is a neighbor of some node j uptree of i. However, this change can only occur due to an update message u sent by j before t_0, contradicting the assumption for u.
3) The Lemma is violated by a message u sent by node j uptree from i at t_o^+. Since $\widehat{W_u} = \widehat{W_j}(t_0^+)$, this would contradict either Lemma 2 or the value of $\lambda_j(\widehat{W_{P_j}})$, for which we have shown in 2) that the Lemma holds.
4) The Lemma is violated indirectly due to a node k changing its next hop towards a node j uptree from i. In this case: $\widehat{W_k}(t_0^+) = W_k(t_0^+) > \lambda_k(W_j)(t_0^+) = \lambda_k(\widehat{W_j})(t_0^+) > W_0$, where W_0 is either W_i or $\widehat{W_i}$ according to the induction hypothesis with respect to $IsAck(i)$. Since we proved in 1) that the Lemma holds for any node k that encounters an event with respect to itself, it must also hold with respect to i. A contradiction.

B. Convergence of SF

Proof of Lemma 4

Let t_1 be a time by which all messages that were sent before t_0 have reached their destination or were dropped. For every $t > t_1$, $IsAck(j)$ changes to $false$ for some node j only due to the receipt by j of an update message u with $T_u = 0$, from a node $i = P_j$. Since $T_j(t^-) = 1$, when i sent u, $IsAck(i)$ must have changed to $false$ and will remain so because i has not received an Ack from j. Therefore, it follows from Lemma 3 that: $\lambda_j(\widehat{W_i})(t^-) \geq W_i(t^-) = W_i(t^+)$. The proof is completed by observing that $W_j(t^-) > \lambda_j(W_i)(t^-) = \lambda_j(\widehat{W_i})(t^-)$ and W_j does not change at t^+.

Proof of Lemma 5

Assume that $IsAck(i) = false$ forever. Therefore, some neighbor j did not send i an acknowledgement. This can happen only if $P_j = i$ and $IsAck(j) = false$. Since the graph is finite and there are no cycles, applying the same argument for j and its uptree nodes produces a contradiction.

Proof of Lemma 6
According to Lemmas 4 and 5, the minimum value of W over nodes for which $IsAck = false$ after t_1 increases with time. Following the same line of proof as [8], we claim that W can either be ∞ or have one of a finite number of possible values, because the cardinality of any parameter that W depends on is finite. Therefore, there will be no more transitions of $IsAck$ to $false$ in finite time. Lemma 5 guarantees that there exists a time t_2 for which all remaining nodes with $IsAck = false$ will change to $true$.

C. Correctness of MV

*Proof of Lemma*7
Once there are no vote or link changes, the effects of the last tree expansion wave will expire in finite time because there are no loops and the graph is finite. Once this happens, no node will change its next hop unless it is inactive. As there are no topological changes, it follows from the SF algorithm that a node can be inactivated only if the node's tree root is inactive.

*Proof of Lemma*8
Let i be a charged root that establishes an inverse hop representing a path to a (nearby) charged root j with an opposite sign and a higher ID. Assuming that j is not neutralized, at the time that i establishes the inverse hop there exists a path from i to j over tree edges of the two roots' trees and the inverse hop edge connecting those trees. Since messages are delivered in FIFO order and the algorithm transfers the charge in step (1) before calling $UnRoot$ in step (2), the charge will traverse each inverse hops before it is deleted. It will then follow next hops of j's nodes (which are not changed) and fuse with j's charge in finite time.

*Proof of Lemma*9
Since fused charges are never separated and no new charges are introduced to the system, the number of distinct charges (whether counted as roots or transfer messages) is positive and a nonincreasing function of time. Therefore, there exists a time t_2 s.t. for every $t > t_2$ this function has a constant value N. Let $C_k, k \in \{1...N\}$ denote the remaining charges after t_2 in decreasing ID order. If C_1 is in transit, it will turn into a charged root in finite time since all nodes marked with a higher ID will either be inactivated or join active trees with a lower ID. (Any previously charged root with a higher ID must have been Unrooted, and *SF* guarantees that its nodes will be inactivated in finite time. In the meantime, C_1 can travel but no higher-ID roots will be created.) Once C_1 establishes a charged root, it will never be neutralized. Now examine C_2. If it has the same sign as C_1, the same logic applies. If C_2 has the opposite sign, it can eventually follow only the tree edges of C_1's root. From Lemma 8 and the fact that the number of charges is constant, it follows that C_2 will establish a permanent root in finite time. Using the same line of arguments, we conclude that all charges establish permanent roots in finite time, thereby guaranteeing termination.

*Proof of Lemma*10
Let i and j be charged roots of neighboring trees of opposite signs. Assume $TreeID_i > TreeID_j$. Examine a path along j's tree connecting j and a node k in j's tree, which has a neighbor from i's tree. Each node along this path from k to j, either has an inverse hop towards i, or towards another root i' with an opposite sign and a higher ID than j's. In any case j must transfer its charge, thus contradicting termination.

*Proof of Lemma*11
The following equation holds for every node i at all times:

$$\sum_{j \in N^i} C_i(j) + C_i = \lambda_d Y_i - \lambda_n V_i$$

(On vote changes, $\Delta\lambda_d$ is added to both sides. In every other event charge is only transferred between the terms on the left.) Therefore, for every group of nodes *X* at all times:

$$\sum_{i,j \in X} C_i(j) + \sum_{i \in X} C_i = \lambda_d \sum_{i \in X} Y_i - \lambda_n \sum_{i \in X} V_i$$

If no transfers are underway, then for every two neighbors i, j : $C_i(j) = -C_j(i)$. If *X* forms a connected component, then the term $\sum_{i,j \in X} C_i(j)$ evaluates to zero, yielding the expected result.

D. Locality properties of MV

*Proof of Lemma*12
Without loss of generality, assume that all trees in E are positive. Let j be a charged root of such a tree, and farther than $2R$ from i. Denote by $Tree_x$ all nodes uptree from some root x. For any node k in $Tree_j$, $d(k, i) > d(k, j)$. This means that k can join $Tree_i$ only after $Tree_j$ is inactivated, and k's weight is increased.

As discussed in section III-A, inactivating a tree takes at least twice its maximum radius (accounting for the inactivation and acknowledgement phases). On one hand, the length of any expansion path of $Tree_i$ through nodes that belonged to $Tree_j$ is at most $Tree_j$'s diameter. On the other hand, $Tree_i$ is stalled at least by $Tree_j$'s diameter before it can expand into $Tree_j$. Therefore, $Tree_i$ requires at least twice the network delay to increase its radius through nodes of $Tree_j$. Since E is totally covered by positive trees we have the result.

*Proof of Theorem*4
Initially, all trees in the graph have the same sign. (If there are no trees, we define R as the graph diameter so the proof is trivial.) If the vote change is one that brings about a change in the majority decision, the problem is known to be global and convergence time depends on the diameter of the graph. We therefore only consider the case in which the vote change does not alter the majority.
Without loss of generality, consider positive roots, i.e., the majority decision is $true$. (We know that all roots have the same sign.) Let i be a node that changed a vote, and let j be the root of the tree to which i belongs. ($i = j$ is possible.) If i's charge remained or changed to positive, i will either create a new root or adjust the ID of an existing root in $O(R)$ time steps. If i's load is now negative, there are two possibilities:

1) i's new ID is lower than $Tree_j$'s ID, and $i \neq j$. In this case the new negative load will be routed to j. According to our optimized ID policy, j's charge is at least as high as i's. Therefore, j will remain positive or cancel itself. In either case the algorithm will adjust in $O(R)$ time steps.
2) i will start creating a new tree. As long as i's ID is higher than the IDs of neighboring trees, i will remain an active root. Since i's new charge resulted from a single vote change (a change in several votes or in i's voting power is proven similarly), the absolute value of i's charge is at most λ_d. Therefore, after at most $O(\lambda_d R) = O(R)$ time steps, enough positive charges will have been routed to cover i or to cause i's negative charge to be routed to a positive charge with a higher ID as in the previous case. Once i's charge is non-negative, this information will be propagated along $Tree_i$ at full speed. Following Lemma 12, since $Tree_i$'s expansion propagates with at most half this speed, all nodes with a negative tree state will either flip their sign or be inactivated in $O(log(\lambda_d R)) = O(log(R))$ time steps. Finally, since there are no remnants of the negative trees, the algorithm will converge in O(R) additional steps.

*Proof of Theorem*5
Without loss of generality, assume that all the existing charged roots are positive. After R time steps, all messages carrying a set expansion flag die out, since none of the new trees can initially expand farther than R before encountering nodes with lower weights. Following Lemmas 7,8 and 9, there exists a time t_2 s.t. for any $t > t_2$, the number of distinct *negative* charges is constant. Therefore, no more fusions take place. Assume that there are still N negative charges after this time. Let $C_k, k \in \{1...N\}$ denote the remaining negative charges in decreasing ID order. Consider C_1. After a finite time, any node whose tree sign is positive and whose tree ID is higher than C_1, is part of an active positive tree that is never unrooted. Therefore, if C_1 is in transit it would eventually fuse with a positive charge, since it can only traverse positive nodes with a higher ID. We conclude that C_1 must become a static charged root in finite time. Now apply the same argument for positive charges whose IDs fall in the range (C_1, C_2), then for C_2 and so on. Hence all charges become static in finite time. If there are still negative charges, eventually two trees with opposite signs will collide, resulting in the transfer of one of the static. This contradicts the assumption that there are remaining negative charges after t_2. Finally, following the same line of proof as step 2 of Theorem 4, all remnants of negative trees will disappear, guaranteeing convergence in finite time.

References

[1] N. Linial, "Locality in distributed graph algorithms," *SIAM J. Computing*, vol. 21, pp. 193–201, 1992.
[2] B. Awerbuch, A. Bar-Noy, N. Linial, and D. Peleg, "Compact distributed data structures for adaptive network routing," pp. 230–240, May 1989.
[3] S. Kutten and D. Peleg, "Fault-local distributed mending," August 1995.
[4] S. Kutten and B. Patt-Shamir, "Time-adaptive self-stabilization," pp. 149–158, August 1997.
[5] R.Wolff and A. Schuster, "Association rule mining in peer-to-peer systems," *In Proc. of the IEEE Conference on Data Mining (ICDM)*, November 2003.
[6] L.R. Ford and D.R. Fulkerson, *Flows in Networks*, Princton University Press, 1962.
[7] J.M. Jaffe and F.H. Moss, "A responsive routing algorithm for computer networks," *IEEE Transactions on Communications*, pp. 1758–1762, July 1982.
[8] J.J. Garcia-Luna-Aceves, "A distributed, loop-free, shortest-path routing algorithm," pp. 1125–1137, June 1988.

[9] F. Kaashoek and D. Karger, “Koorde: A simple degree-optimal distributed hash table,” February 2003.

[10] R. Govindan C. Intanagonwiwat and D. Estrin, “Directed diffusion: A scalable and robust communication paradigm for sensor networks,” August 2000.

[11] David Kempe, Alin Dobra, and Johannes Gehrke, “Computing aggregate information using gossip,” 2003.

[12] Mayank Bawa, Hector Garcia-Molina, Aristides Gionis, and Rajeev Motwani, “Estimating aggregates on a peer-to-peer network,” Tech. Rep., Stanford University, Database group, 2003, Available from: `http://www-db.stanford.edu/~bawa/publications.html`.

Algorithm 1 Spanning Forest (SF)

```
Variables for node i:
  • $R_i, T_i, W_i, A_i, P_i$ - Root and tree activity states {0,1}, path weight and Ack number (positive Int), and a next hop pointer, respectively.
  • $\forall j \in N^i : \lambda_i(T_j), \lambda_i(W_j), \lambda_i(A_j)$ - A neighbor j's tree state, weight and Ack as known to i.
Macros:
$Inactive(i) \;\equiv\; (T_i = 0) \bigvee (P_i \neq \perp \bigwedge \lambda_i(T_{P_i}) = 0)$
$IsAck(i)$ - Evaluates to $true$ iff $i$'s neighbors have all acknowledged $i$'s most recent (highest) Ack number. New neighbors are considered to have sent and received all Acks that could have been pending to or from their neighbors.
(The details of ack management are omitted for brevity, but are included in the running code.)

Events: /* trigger + event specific action */
  • $Init_i() : R_i = 0, T_i = 0, W_i = \infty, P_i = \perp, A_i = 0,$
    $\forall j \in N^i : LinkDown_i(j)$.
  • $LinkUp_i(j)$: send $Update(T_i, W_i, A_i)$ to $j$.
  • $LinkDown_i(j) : \lambda_i(T_j) = 0, \lambda_i(W_j) = \infty,$
    $\lambda_i(A_j) = \perp$. if $(P_i = j)$ $P_i = \perp$.
  • $Root_i$ operation: $R_i = 1$.
  • $UnRoot_i$ operation: $R_i = 0$.
  • receive $Update(T, W, A)$ from $j$:
    update $\lambda_i(T_j)$, $\lambda_i(W_j)$, and $\lambda_i(A_j)$.
  • receive $Ack(A)$ from $j$: record the most recent version of $i$'s Ack number acknowledged by $j$.
After every event also do: /* common actions */
  1) if $R_i = 1$: /* set $i$ as an active root: */
     a) $T_i = 1, W_i = 0, P_i = \perp$
  2) else: /* $R_i = 0$ */
     a) /* if $i$ is inactive and all uptree nodes have acknowledged, update $i$'s weight according to its next hop: */
        if ($T_i = 0 \bigwedge IsAck(i) = true$)
            $W_i = \begin{cases} \infty, & P_i = \perp \bigvee \lambda_i(W_{P_i}) = \infty \\ \lambda_i(W_{P_i}) + d(i, P_i), & \text{otherwise} \end{cases}$
     b) /* improve $i$'s path or join an active tree with the same weight if $i$ is inactive or about to become inactive: */
        let $j \in N^i$ s.t. $W(j)$ is minimal, where
        $W(j) = \begin{cases} \lambda_i(W_j) + d(i, j), & \lambda_i(T_j) \neq 0 \\ \infty, & \text{otherwise} \end{cases}$
        if $(W(j) < W_i \;\bigvee$
              $(W(j) = W_i \;\bigwedge\; W(j) < \infty \;\bigwedge\; Inactive(i)))$
                $P_i = j, W_i = \lambda_i(W_j), T_i = \lambda_i(T_j)$
     c) /* if $i$ is turning inactive, increment $i$'s Ack: */
        if $(T_i \neq 0 \;\bigwedge\; (P_i = \perp \bigvee \lambda_i(T_{P_i}) = 0))$
            $T_i = 0, A_i = A_i + 1$
  3) send $Update(T_i, W_i, A_i)$ to all neighbors if something changed.
  4) send $Ack(\lambda_i(A_j))$ to each unacknowledged neighbor $j$, with the exception of $P_i$ if $IsAck(i) = false$.
The answer to the $NextHop_i$ query is $P_i$'s current value.
```

Algorithm 2 Majority Vote

Variables for node *i*:

- Y_i, V_i, C_i, ID_i - "Yes" votes, total votes,charge and charge ID, respectively.
- $\forall j \in N^i : C_i(j)$ - total charge transferred **between** *i* and a neighbor *j*, from i's perspective.

Macros:

$GenID(charge)$ generates a new charge ID
$Charge(V, Y) = \lambda_d \cdot Y - \lambda_n \cdot V$

Events: /* trigger + event specific action */

```
• $Init_i$: $V_i, Y_i, C_i = Charge(V_i, Y_i)$,
  $ID_i = GenID(C_i), \forall j \in N(i) : C_i(j) = 0$.
• $LinkUp_i(j)$: do nothing.
• $LinkDown_i(j)$: $C_i += C_i(j), C_i(j) = 0$.
• $ChangeVote_i(V, Y)$:
  $C_i += (Charge(V, Y) - Charge(V_i, Y_i))$,
  $V_i = V, Y_i = Y, ID_i = GenID(C_i)$.
• Receive $Transfer(C, ID)$ from $j$:
  if ($C_i = 0$) /* i is currently neutral */
     $ID_i = ID$
  else /* fusion is taking place - update charge id */
     $ID_i = GenID(C_i + C)$.
  $C_i += C, C_i(j) -= C$.
/* common actions */
After each of the events above or an event in SF do:
  1) /* if $i$ is charged, try to transfer the charge: */
     if ($C_i \neq 0 \wedge TreeID_i \geq ID_i$)
        if ($Sign(C_i) = -TreeSign_i$)
           $temp = NextHop_i$ else $temp = InvHop_i$.
        if ($temp \neq \perp$) send $Tansfer(C_i, ID_i)$ to $temp$,
           $C_i(j) += C_i; C_i = 0$.
  2) /* if $i$ remained charged, verify it is marked as an active
     root. Otherwise, unmark it: */
     if ($C_i = 0$) $UnRoot_i$ else $Root_i(Sign(C_i), ID_i, f)$
     where $f = true$ if invoked by a $ChangeVote_i$
     operation.
Output: $true$ if $TreeSign_i \geq 0$, and $false$ otherwise.
```