CCIT Report #590 May 2006

# A Local Algorithm for Ad Hoc Majority Voting Via Charge Fusion

Yitzhak Birk, Liran Liss, Ran Wolff and Assaf Schuster
*Technion – Israel Institute of Technology*
{birk@ee,liranl@tx,ranw@cs,assaf@cs}.technion.ac.il

June 24, 2006

**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 one 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.

# 1 Introduction

## 1.1 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 effectively independent of the graph size. Examples include Ring Coloring [1] and Maximum 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 [3]. Another example is fault-local mending algorithms [4, 5]. 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 [4, 5] 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. A non-trivial measure with a low value for many instances suggests 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.

## 1.2 The Majority Vote Problem

Consider a system comprising an unbounded number of interconnected nodes. 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 on system size. In certain cases such as a tie, computing the majority would require collecting at least half of the votes, which would indeed take time proportional to the size of the system. Yet, it appears possible that whenever the

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

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 quickly squelch the wave of erroneous perceived outcome, 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 regarding the majority on the sign of its charge or on 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.

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 [6], 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. Section 2 provides an overview of our approach. Section 3 presents a basic spanning forest (SF) algorithm, which is adapted to routing charges in section 4. Our majority vote (MV) algorithm is detailed in section 5. The correctness and locality properties of MV are proved in sections 6 and 7, respectively. In section 8, we provide some empirical results to confirm our assumptions and demonstrate the performance of our algorithm. Section 9 describes at some length related work. Section 10 offers concluding remarks.

## 2 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 poll balanced. Every balanced poll bases its current belief regarding the majority 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 at any time, we must distinguish between a node's current vote and the votes or "tokens" that are transferred 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 majority vote decision allows us to work with integers and only deal with linear operations (addition and subtraction):

$$\text{Yes} \Leftrightarrow \lambda_d \sum_{i \in X} Y_i - \lambda_n \sum_{i \in X} V_i \geq 0.$$

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 $\lambda_d$. An addition of one vote to the voting power of a node reduces its charge by $\lambda_n$ 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 can transfer charge such that *nearby* charged nodes with opposite signs cancel one another

without introducing a livelock, and subsequently disseminate the resulting majority decision to neutral nodes locally, we will 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, and SF guarantees that all neutral nodes join trees rooted at remaining charged ones in their vicinity. Each node bases its (perceived global) majority decision on the sign of the charge of its tree's root. Therefore, disseminating a majority decision to all nodes is inherently built into the algorithm.

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.

# 3 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 for use by MV.

## 3.1 SF Algorithm description

Given an undirected graph with positive integer edge weights 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. 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 formally states SF. In addition to topological changes, the algorithm supports two operations that 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 [7]: 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 2). 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 [8]. (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 (or that leave $i$'s weight unchanged when it would be inactive otherwise), and by allowing $i$ to increase its current weight only when $i$ and all its uptree nodes are inactive (step 1).

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 6), before $i$ is itself inactivated ($T_i$ is set to 0 and $A_i$ is incremented in step 4) 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 its tree as inactive and reset their weight to $\infty$. 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 $\infty$. This weight increase propagates towards the leaves, resetting the weight of all nodes in the tree to $\infty$ 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 [8].)

## 3.2 Loop Freedom

We prove that SF is loop-free at every instant by carefully reasoning about possible node weights in trees. We say that a node $i$ is *active* if $T_i = 1$, and *inactive* if $T_i = 0$. For facility

**Algorithm 1** Spanning Forest (SF)

**Definitions**: $N^i$ is the set of $i$'s neighbors

**Variables for node i**:

- $R_i \in \{0,1\}$ (root activity state), $T_i \in \{0,1\}$ (tree activity state), $W_i \in \mathbb{N}$ (path weight), $A_i \in \mathbb{N}$ (Ack number), $P_i \in N^i \cup \{\perp\}$ (next hop pointer)
- Denote by $\chi = \{T, W, A\}$ the set of *visible* variables; $\forall j \in N^i, X \in \chi$: $\lambda_i(X_j)$ holds the value of $X_j$ as known to $i$.

**Macros:**

- $Inactive(i) \equiv (T_i = 0) \vee (P_i \neq \perp \wedge \lambda_i(T_{P_i}) = 0)$ - Evaluates to *true* iff $i$ is inactive or its current next hop is assumed to be inactive
- $IsAck(i)$ - Evaluates to *true* iff $i$'s neighbors have all acknowledged $i$'s most recent (highest) Ack number. Nodes that become neighbors are considered to have sent and received all Acks that could have been pending to or from each other. (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 $i$ is inactive and all uptree nodes have acknowledged, update $i$'s weight according to its next hop: */
   if $(T_i = 0 \wedge IsAck(i) = true)$
   $$W_i = \begin{cases} \infty, & P_i = \perp \vee \lambda_i(W_{P_i}) = \infty \\ \lambda_i(W_{P_i}) + d(i, P_i), & \text{otherwise} \end{cases}$$
2. /* improve $i$'s path: */
   let $j \in N^i$ s.t. $W_i(j)$ is minimal, where
   $$W_i(j) = \begin{cases} \lambda_i(W_j) + d(i,j), & \lambda_i(T_j) \neq 0 \\ \infty, & \text{otherwise} \end{cases}$$
   if $((W(j) < W_i) \vee (W(j) = W_i \wedge W(j) < \infty \wedge Inactive(i)))$
   $P_i = j, W_i = W_i(j), T_i = \lambda_i(T_j)$
3. /* if $i$ was marked as a root, set variables accordingly: */
   if $(R_i = 1)$
   $T_i = 1, W_i = 0, P_i = \perp$
4. /* if $i$ is turning inactive, increment $i$'s Ack: */
   if $(T_i \neq 0 \wedge ((P_i = \perp \wedge R_i = 0) \vee (P_i \neq \perp \wedge \lambda_i(T_{P_i}) = 0)))$
   $T_i = 0, A_i = A_i + 1$
5. send $Update(T_i, W_i, A_i)$ to all neighbors if something changed.
6. send $Ack(\lambda_i(A_j))$ to each unacknowledged neighbor $j$, with the exception of $P_i$ if $IsAck(i) = false$.

**Output:** The $NextHop_i$ query returns $P_i$'s current value.

of exposition, we also make the following definitions:

**Definition 3.1** (generalized weight)**.** *The* generalized weight[2] *of a node $i$, $\widehat{W}_i$, equals $W_i$ if $i$ is active and $\infty$ otherwise.*

**Definition 3.2** (dead branch node)**.** *$i$ is a* dead branch *node if the following conditions hold for every node $j$ uptree from or equal to $i$:*

1. $T_j = 0$.
2. *For every in-flight update message $u$ sent by $j$:* $T_u = 0$.
3. *For every neighbor $m$ of $j$:* $\lambda_m(T_j) = 0$.
4. *If $j \neq i$:* $IsAck(j) = true$.

[2]This notion is only used in proofs.

We begin by stating two simple facts, and establishing sufficient conditions for determining dead-branch nodes. The proofs are deferred to Appendix A.

**Lemma 3.3.** *After any event in which $\widehat{W}_i$ increases for some nonisolated node $i$, $\widehat{W}_i = \infty$, $T_i = 0$ and $IsAck(i) = false$.*

**Lemma 3.4.** *If $P_i \neq \perp$, then either $\widehat{W}_i > \lambda_i(\widehat{W}_{P_i})$ or $\widehat{W}_i = \lambda_i(\widehat{W}_{P_i}) = \infty$.*

**Lemma 3.5.** *If one of the following holds for some node $i$ at time $t$:*

1. *$T_i = 0$ and $IsAck(i) = true$.*
2. *$T_i = 0$, $IsAck(i) = false$, and all of $i$'s pending acknowledgements are in-flight.*

*then $i$ is a dead branch node.*

We now show that the generalized weights of nodes uptree from $i$ or messages sent by them are at least as high as $i$'s, depending on $i$'s acknowledgement status. Loop-freedom follows.

**Lemma 3.6.** *For every node $i$*

1. *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. *if $IsAck(i) = false$, then the same claims hold when replacing $\widehat{W}_i$ with $W_i$.*

*Proof.* Initially all nodes are isolated so the Lemma holds trivially. Consider an event at time $t$, and assume that the Lemma was correct with respect to some non-isolated node $i$ at $t^-$ (a reconnecting isolated node has no uptree nodes and no messages in flight). We show that the Lemma still holds at $t^+$ by contradiction. The Lemma can be violated in the following cases:

(a) The event occurs in $i$. Since for any update message $u$ sent by $i$ at $t^+$: $\widehat{W}_u = \widehat{W}_i(t^+)$, 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:

- $IsAck(i)(t^-) = IsAck(i)(t^+) = true$. Lemma 3.3 guarantees that $\widehat{W}_i$ cannot increase. Therefore, 1) cannot be violated and 2) does not apply.
- $IsAck(i)(t^-) = IsAck(i)(t^+) = false$. Since $W_i$ can only increase in step 1, and $IsAck(i)(t^-) = false$, $W_i$ cannot increase. Therefore, 2) cannot be violated.
- $IsAck(i)$ changes from $true$ to $false$. Since this change can only occur if $T_i(t^-) = 1$, it follows from the algorithm that $W_i$ cannot change, and we have: $\widehat{W}_i(t^-) = W_i(t^-) = W_i(t^+)$. Therefore, 2) holds at $t^+$ due to our assumption that 1) held at $t^-$.

- $IsAck(i)$ 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 $T_i(t^-) = 1$, it follows from the algorithm that $W_i$ cannot change, and we have: $W_i(t^-) = W_i(t^+) = \widehat{W}_i(t^+)$. Therefore, 1) holds at $t^+$ due to our assumption that 2) held at $t^-$. If $T_i(t^-) = 0$, according to Lemma 3.5 $i$ is a dead branch, so 1) holds trivially at $t^+$.

(b) 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 $t$, contradicting the assumption for $u$.

(c) The Lemma is violated by a message $u$ sent by node $j$ uptree from $i$ at $t^+$. Since $\widehat{W}_u = \widehat{W}_j(t^+)$, this would contradict either Lemma 3.4 or the value of $\lambda_j(\widehat{W_{P_j}})$, for which we have shown in (b) that the Lemma holds.

(d) The Lemma is violated indirectly due to a node $k$ changing its next hop towards a node $j$ uptree from $i$. In this case: $\widehat{W}_k(t^+) = W_k(t^+) > \lambda_k(W_j)(t^+) = \lambda_k(\widehat{W}_j)(t^+) > W_0$, 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 (a) 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. ■

**Theorem 3.7.** *There are no cycles in the graph at any instant.*

*Proof.* Let $i$ be a node that closes a cycle at time $t$. Therefore, at $t^+$ we have $\lambda_i(\widehat{W}_{P_i}) = \lambda_i(W_{P_i}) < W_i = \widehat{W}_i$. However, since $P_i$ is also uptree from $i$, it follows from Lemma 3.6 that $\lambda_i(\widehat{W}_{P_i}) \geq \widehat{W}_i$, a contradiction. ■

### 3.3 Convergence

We say that SF is converged if no messages are sent, all nodes are acknowledged ($IsAck =$ *true* for all nodes), and one of the following conditions hold:

- If no node is marked as an active root, then all nodes in the graph are inactive.
- Otherwise, all nodes are active and belong to trees with marked roots.

**Theorem 3.8.** *SF converges within finite time after* $t_0$.

The proof, which follows from the properties of the standard Bellman-Ford algorithm, is deferred to Appendix A.

## 4 Adapted Spanning Forest Algorithm

In order to use SF for routing charges, we inserted several changes to algorithm 1. The adapted spanning forest algorithm (ASF) is algorithm 2. (New functional sections that were added to the SF algorithm are preceded by ASF remarks.) We next detail each of these changes.

## **Algorithm 2** Adapted Spanning Forest (ASF)

**Variables for node i**:

- SF variables:
  $R_i \in \{-1,0,1\}$ (root activity state), $T_i \in \{-1,0,1\}$ (tree activity state), $W_i \in \mathbb{N}$ (path weight), $A_i \in \mathbb{N}$ (Ack number), $P_i \in N^i \cup \{\perp\}$ (next hop pointer)
- Additional ASF variables:
  $D_i \in \mathbb{N}$ (delay), $RID_i \in \mathbb{N}$ (root ID), $TID_i \in \mathbb{N}$ (tree ID), $\overline{P}_i \in N^i \cup \{\perp\}$ (inverse hop pointer), $\overline{W}_i \in V$ (inverse hop wieght), $EP_i \in N^i \cup \{\perp\}$ (expansion source pointer), $EL_i$ (expansion time to live)
- $\forall j \in N^i : \Delta_i(j) \in \mathbb{N}$ - $i$'s delay counter for neighbor $j$.
- Denote by $\chi = \{T, TID, P, W, D, \overline{W}, A\}$ the set of *visible* variables; $\forall j \in N^i, X \in \chi$: $\lambda_i(X_j)$ holds the value of $X_j$ as known to $i$.

**Macros and constants:**

- $IsAck(i)$ - as in SF.
- $Inactive(i)$ - as in SF.
- $\alpha > 1$ - Max permissible ratio between $D_i$ and $W_i$. Typically $\alpha = 4$.
- $CanChange(i) \equiv (T_i = 0 \wedge IsAck(i) = true)$ - node $i$ may change $P_i$ only if it is inactive and acknowledged.
- $CanRoot(i) \equiv (EP_i = i) \vee$
  $((T_i = 0 \wedge IsAck(i) = true) \vee (T_i = 1 \wedge P_i = \perp))$ - node $i$ may become a root (with a new tree ID) either if it is inactive and acknowledged or already an active root.
- $IsDelay(i,j) \equiv (\lambda_i(D_j) + \Delta_i(j) > \lambda_i(W_j) + d(i,j))$.

**Events:** /* trigger + event-specific action */

- $Init_i()$ : $R_i = 0, T_i = 0, W_i = \infty, D_i = 0, P_i = \perp, A_i = 0, RID_i = \infty, TID_i = \infty, \overline{P}_i = \perp, \overline{W}_i = \infty, EP_i = \perp, EL_i = 0,$
  $\forall j \in N^i : LinkDown_i(j)$.
- $LinkUp_i(j)$: send $Update(\chi_i, 0)$ to $j$.
- $LinkDown_i(j)$: if $(P_i = j)$ $P_i = \perp$. $\forall X \in \chi_i$ : set $\lambda_i(X)$ to $X$'s initial value.
- $LinkWeightChange_i(j, W)$:
  Set estimate for $d(i,j)$ as $W$.
- $Root_i(R, RID, expand)$ operation:
  $R_i = R, RID_i = RID$.
  if $(expand = true)$ $EP_i = i$,
  if $(W_i < \infty)$ $EL_i = W_i$.
- $UnRoot_i$ operation: $R_i = 0$.
- receive $Update(\chi, EL)$ from $j$:
  if $(\lambda_i(T_j) = 0 \vee \lambda_i(W_j) \neq W)$ $\Delta_i(j) = 0$;
  $\forall X \in \chi : \lambda_i(X_j) = X$;
  $EL_i = max(0, EL - d(i,j))$;
  if $(EL_i > 0)$ $EP_i = j$.
- receive $Ack(A)$ from $j$: record the value of $i$'s Ack num as acknowledged by $j$.
- on $ClockTick$: $\forall j \in N^i$ :
  if $(\lambda_i(T_i) \neq 0 \ \wedge \ \lambda_i(D_j) + \Delta_i(j) < \alpha(\lambda_i(W_j) + d(i,j)))$
  $\Delta_i(j) = \Delta_i(j) + 1$.

**After every event also do:** /* common actions */

1. /* ASF: Update sign and ID changes in existing trees */
   if $(T_i \neq 0 \wedge P_i \neq \perp \wedge \lambda_i(T_{P_i}) \neq 0)$
   $T_i = \lambda_i(T_{P_i}), TID_i = \lambda_i(TID_{P_i})$
2. /* if $i$ is inactive and all uptree nodes have acknowledged, update $i$'s weight according to its next hop: */
   if $(T_i = 0 \wedge IsAck(i) = true)$
   $$W_i = \begin{cases} \infty, & P_i = \perp \vee \lambda_i(W_{P_i}) = \infty \\ \lambda_i(W_{P_i}) + d(i,P_i), & \text{otherwise} \end{cases}$$
3. /* improve $i$'s path */
   let $j \in N^i$ s.t. $W_i(j)$ is minimal, where
   $$W_i(j) = \begin{cases} \lambda_i(W_j) + d(i,j), & \lambda_i(T_j) \neq 0 \wedge \\ & (EP_i = j \vee \\ & ((j = P_i \vee \\ & \quad CanChange(i)) \wedge \\ & \quad IsDelay(i,j))) \\ \infty, & \text{otherwise} \end{cases}$$
   if $((W_i(j) < W_i) \vee$
   $(W_i(j) = W_i \wedge W_i(j) < \infty \wedge Inactive(i)))$
   $P_i = j, W_i = W_i(j),$
   $T_i = \lambda_i(T_j), TID_i = \lambda_i(TID_j),$
   $D_i = min(\lambda_i(D_j) + \Delta_i(j),$
   $\alpha(\lambda_i(W_j) + d(i,j)))$
4. if $(R_i \neq 0 \wedge CanRoot(i))$
   $T_i = R_i, TID_i = RID_i,$
   $W_i = 0, D_i = 0, P_i = \perp$
5. /* if $i$ is turning inactive, increment $i$'s Ack: */
   if $(T_i \neq 0 \ \wedge \ ((P_i = \perp \wedge R_i = 0) \vee$
   $(P_i \neq \perp \wedge \lambda_i(T_{P_i}) = 0)))$
   $T_i = 0, A_i = A_i + 1$
6. /* ASF: update inverse hop information: */
   if $(T_i = 0)$ $\overline{W}_i = \infty$ else
   let $j \in N^i$ s.t. $\overline{W}(j)$ is minimal, where
   $$\overline{W}(j) = \begin{cases} \lambda_i(\overline{W}_j) + d(i,j), \ \lambda_i(TID_j) = TID_i \wedge \\ \quad \lambda_i(P_j) = P_i \\ \lambda_i(W_j) + d(i,j), \ \lambda_i(TID_j) > TID_i \wedge \\ \quad \lambda_i(T_j) = -T_i \\ \infty, \quad \text{otherwise} \end{cases}$$
   $\overline{W}_i = \overline{W}(j), \overline{P}_i = j$
7. if $(\overline{W}_i = \infty)$ $\overline{P}_i = \perp$
8. if $(EP_i = P_i)$ $temp = EL_i$ else $temp = 0$;
   send $Update(\chi_i, temp)$ to all neighbors if anything changed.
9. send $Ack(\lambda_i(A_j))$ to each unacknowledged neighbor $j$, with the exception of $P_i$ if $IsAck(i) = false$.
10. /* ASF: reset expansion wave variables: */
    $EP_i = \perp; EL_i = 0$

**Output:** The $NextHop_i$, $InvHop_i$ and $TreeSign_i$ queries return the current values of $P_i$, $\overline{P}_i$ and $T_i$, respectively.

## 4.1 Expanding the tree state to include sign information

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.

## 4.2 Associating an ID with every tree

We attach "Root ID" ($RID$) and "Tree ID" ($TID_i$) variables to each node for symmetry breaking as explained next. Every active root is assigned an ID during the $Root_i$ operation, which is propagated throughout its tree.

## 4.3 Adding inverse hops

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.

Inverse hops are represented by weight ($\overline{W}_i$) and pointer ($\overline{P}_i$) variables. The foregoing logic for determining a node's inverse hop is specified in steps 6 and 7 of ASF. Note that only active nodes can have inverse hops.

## 4.4 Limiting tree expansion and path improvements

To guarantee that paths do not break while routing charges, we normally allow only a dead-branch node to change its next hop. However, as will be explained in the next section, there are cases in which new active roots should be able to take over active nodes of neighboring trees. Therefore, we extend the *Root* operation to include an *expansion flag*. Setting this flag creates a bounded one-shot expansion wave, by repeatedly allowing any neighboring nodes to join the tree. The wave will die down when it stops improving the shortest path of neighboring nodes or when the bound is reached.

Expansion waves are realized by adding two new variables: an expansion time-to-live ($EL_i$) and a pointer ($EP_i$). When the $Root_i$ operation is invoked with the expansion flag set, $EL_i$ is set equal to the node's current weight (if it is finite), and $EP_i$ is set to $i$ to signify that an expansion wave was initiated in $i$. Any update message that $i$ sends (step 8) will convey $EL_i$'s value.

If $i$ is a node that receives a message with a positive $EL$ value from a neighbor $j$, it decrements this value (by the delay of the link from which the message was received) and sets $EL_i$ accordingly. If $EL_i$ is positive, $EP_i$ is set to $j$ to signify that an expansion wave had been accepted from $j$. In this case, $i$ can change its next hop pointer to $j$ even if it is not a dead-branch node, as is evident from step 3 of ASF and the $CanChange$ predicate. If $i$ indeed changes its next hop during this event, then $i$ takes part in the expansion wave and every message it sends will also convey $EL_i$'s value.

Note that the values of $EL_i$ and $EP_i$ are not not persistent between events, but rather are reset each time in step 10 of ASF. Therefore, expansion waves have limited duration and influence: they are initiated by a *Root* operation with a set expansion flag, carried by messages with positive $EL$ values, and die down when no such messages are sent. Trivially, an expansion wave that originates in $i$ can only reach a distance that is equivalent to $i$'s weight before the wave was initiated (roughly $i$'s distance to its previous root).

## 4.5 Delaying root creation

Due to similar considerations as in the previous case, we delay the creation of a new root at an active non-root node $i$, whenever $Root_i$ is invoked without a set expansion flag. $i$ will, however, become a root immediately after it is inactive and acknowledged unless $UnRoot_i$ was invoked in the meantime. This behavior is enforced in step 4 of ASF and the $CanRoot$ predicate.

## 4.6 Introducing delays

Finally, we slow down the rate at which inactive nodes join active trees by introducing a clock event ($ClockTick$), a delay variable ($D_i$), and a delay counter for every neighbor ($\Delta_i(j)$). Delays are essential to guarantee the local operation of the algorithm in some cases. (We will elaborate on this matter when we discuss the locality properties of the algorithm in section 7.)

The delay mechanism works as follows. Denote by $r$ the root of the tree that $i$ belongs to. $D_i$ generally represents the difference between the time passed from the moment that the tree was created in $r$ until $i$ last joined this tree, and the network propagation time along the tree path from $r$ to $i$. Whenever a new tree is created at some node $r$, $D_r$ is set to 0. When a node $i$ hears about a path with a specific weight to $r$ via a neighbor $j$ (by monitoring changes in $\lambda_i(T_j)$ and $\lambda_i(W_j)$), it records $j$'s weight (in $\lambda_i(W_j)$ as well as $j$'s delay (in $\lambda_i(D_j)$) and resets $\Delta_i(j)$ to 0. In every clock event, $i$ increments $\Delta_i(j)$ (up to a predefined limit). If $i$ eventually selects $j$ as its next hop, it also updates $D_i$ to equal $\lambda_i(D_j) + \Delta_i(j)$ (as long as the value does not exceed a predefined limit) in step 3 of ASF. ($D_i = 0$ indicates that both $i$ and all its downtree nodes actively joined $r$'s tree as soon as possible, i.e., without delays.)

The actual condition that enforces delays and slows down tree expansion rate is the $IsDelay$ predicate used in step 3 of ASF. Namely, $i$ will not change its next hop and join a path to $r$ via $j$ if the accumulated delay of this path ($\lambda_i(D_j) + \Delta_i(j)$) is not at least as

Table 1: ASF 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 |
| $NextHop_i$ | 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 |

high as the resulting path weight $(\lambda_i(W_j) + d(i,j))$. (This restriction is temporarily lifted only for expansion waves.) Thus, the rate in which a path can be extended is reduced to at most half the speed of that if delays never occurred. As a result, any information that is passed from $r$ to the tree nodes without delays (such as inactivating the tree) is guaranteed to quickly reach every node in the tree.

This mechanism enables trees that cannot currently expand (e.g., trees that are surrounded by other active trees) to accumulate a "delay credit" by incrementing $\Delta_i(j)$ values of neighboring nodes. Therefore, if a neighboring tree is subsequently inactivated, its nodes can be quickly taken over.

The interface exposed by ASF algorithm to MV is summarized in Table 1. Although ASF introduces considerable changes to SF, its basic operation remains practically the same: after adjusting to additional sign and ID differences unique to ASF (step 1), the algorithm first adjusts the node's weight if it is a dead-branch (step 2), attempts to improve the node's path (step 3), sees if the node should become a root (step 4), and only then checks if the node should be inactivated (step 5). The changes do not invalidate the algorithm's correctness; in Appendix A, we prove that:

**Proposition 4.1.** *ASF is correct and converges within finite time.*

# 5 Majority Vote Algorithm (MV)

MV is an asynchronous reactive algorithm. It operates by expressing local vote changes as charge, relaying charge sign information among neighboring nodes using ASF, 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 ASF), 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 is marked as an active root (using the ASF *Root* operation) with the corresponding sign and charge ID. If the event was a vote change, we also set the root's expansion flag, allowing its tree to take over nearby active nodes via an expansion wave. This is important because a vote change can introduce a new tree with a certain sign in a region of the graph that contains only trees with that sign; the expansion wave balances the sizes of the new tree and its neighboring trees, increasing the locality of future operations (such as charge fusions).

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 ASF *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 ASF $InvHop$ query), and then along next hops of nodes that are part of the positive tree (using the ASF $NextHop$ 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 3 states MV formally. $C_i(j)$ keeps track of the net charge transferred between a node and each of its neighbors. When an edge $(i, j)$ fails, MV adds a charge of $C_i(j)$ to $i$ and a charge of $C_j(i)$ to $j$. This operation effectively cancels out the net charge transfer that ever took place between $i$ and $j$. Thus, the net charge of a connected component is always determined the votes of its own nodes. $GenID(charge)$ can generally 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 same-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 ASF. 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 ASF's $TreeSign$ query). For inactive nodes, we arbitrarily return *true*.

**Remark.** Apart from environmental events that cause a reaction both in ASF and MV (such as $Init_i$, $LinkUP_i$ and $LinkDown$), the algorithms have a mutual influence: MV is invoked after any change in ASF's state (since the answers to the $TreeSign_i$, $NextHop_i$, or

**Algorithm 3** Majority Vote (MV)

**Definitions**: $N^i$ is the set of $i$'s neighbors

**Variables for node i**:

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

**Macros:**
$GenID(C) = 2(|C| - 1) + \frac{1}{2}(1 + sign(C))$
$Charge(V, Y) = \lambda_d \cdot Y - \lambda_n \cdot V$

**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: update charge id */
  $ID_i = GenID(C_i + C)$.
  $C_i += C, C_i(j) -= C$.

/* common actions */
**After each of the above events or a change in SF do:**

1. /* if $i$ is charged, try to transfer the charge: */
   if $(C_i \neq 0)$
   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.

$InvHop_i$ queries could have changed), and ASF's $UnRoot_i$ and $Root_i$ operations are called directly from MV. Consequently, it may seem that an infinite recursion can occur. However, it is not possible: if a node $i$ remains charged, a second call to $Root_i$ by MV does not change $ASF$'s state; if a node $i$ was neutral or MV decides to transfer its charge, it remains neutral and a second call to $UnRoot_i$ by MV does not change ASF's state. Any either case, the invocation loop is terminated.

# 6 MV Correctness

Assume that all external events (link state changes, vote changes, etc.) stop at some time $t_0$. We say that MV is *correct*, if no more messages are sent within finite time after $t_0$, and the output of every node equals the majority decision in its connected component. (Note that ASF clock events continue after this time, but they do not result in sending new messages.)

Denote by $d_{max}$ the maximum edge delay in the graph at $t_0$. Our correctness proof of the algorithm follows several steps:

1. Establishing some baseline facts that hold within some finite time $t_1$ after $t_0$.
2. Proving several tree properties.
3. Proving that the set of active roots is fixed within finite time.
4. Proving that the algorithm stops within finite time with the correct results.

These steps are detailed in subsequent sections. The following general definitions are used throughout the paper.

**Definition 6.1** (node activation, depth and height)**.** *A node $i$ is said to be* activated *(*inactivated*) at time $t$ if $i$ was inactive (active) at $t^-$ and is active (inactive) at $t^+$. The* depth *of $i$ at time $t$, $Dep_i(t)$, is $i$'s distance from its root in units of network delay. The* height *of $i$ at time $t$, $Height_i(t)$, is $i$'s distance to its farthest uptree node (a leaf) in units of network delay.*

**Definition 6.2** (join event)**.** *A* join event *is an event $e$ in which either (1) an inactive node becomes active, or (2) an active node changes its weight (and remains active). Assume that a join event $e$ occurred at some node $i$. We denote by $t_e$ the time at which $e$ occurred, and by $X_e$ the value of any variable $X_i$ at $t_e^+$ (e.g, $W_e$ stands for $i$'s weight at $t_e^+$). Similarly, we denote $i$'s depth at $t_e^+$ by $Dep_e$.*

A node $i$ is said to *join* node $j$, if it exhibits a join event $e$ such that $P_i(t_e^+) = j$. Note that both $T_e$ and $W_e$ are determined according to the last message $u$ from $j$ that changed either $\lambda_i(T_j)$ or $\lambda_i(W_j)$. However, $e$ does necessarily occur upon receiving $u$. (Since $u$ must have been sent during a previous join event $e'$ at $j$, we also use this terminology with respect to events, i.e., we say that $e$ joined $e'$.)

**Definition 6.3** (event chains and closure)**.** *The* closure *of a join event $e$, $Close(e)$, is a set of join events containing: (1) $e$ itself, (2) any join event $e'$ that joins a previous join event in $Close(e)$. A sequence $\{e_n\}, n \geq 1$ is a* chain *of join events in $Close(e)$, if every event $e_n$ joined $e_{n-1}$ and $e_0 = e$.*

**Definition 6.4** (expansion event)**.** *A join event $e$ at node $i$ is an* expansion event *if one of the following conditions hold:*

1. *A root was created during $e$ due to a vote change at $i$.*
2. *$i$ joined a neighbor $j$ due to an update message $u$ from $j$, such that: (1) $EL_u > d(i, j)$); and (2), $e$ occurred upon receiving $u$.*

*Note that an expansion event enables an active node to change its next hop pointer. Given a vote change event $e$, an* expansion wave *is its closure, $Close(e)$.*

## 6.1 Baseline facts

Say that two charges are *distinct* if they are located in different nodes or carried by different messages. We initially make a simple observation:

**Lemma 6.5.** *The number of distinct charges is constant after some finite time.*

*Proof.* 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 forms a non-increasing function of time. Therefore, after some finite time this function is constant. ■

Unfortunately, this observation alone does not guarantee that all opposite signed charges are fused within finite time because such charges might infinitely chase one another. Therefore, at this stage we only assume that there exists a time after which no more fusions between opposite-signed charges occur. In appendix B, we show that:

**Lemma 6.6.** *As long as no fusions between opposite-signed charges occur, active roots can only increase their IDs.*

**Lemma 6.7.** *All expansion waves die down within finite time.*

Following these results, we conclude that there exists a time by which all expansion waves die down and there are no more fusions between opposite signed charges. Denote this time by $t_1$.

## 6.2 Tree Properties

Assuming that the graph is not the trivial case of an isolated node, we now establish several tree properties that hold after $t_1$. These properties, namely *uniformity*, *stability*, *inverse-stability*, and *full-inactivity*, are later used to prove that all opposite-signed charges in the system are fused within finite time. Another related concept that characterizes the graph as a whole rather than specific trees is a *stable ID*.

Since trees are solely managed by ASF, any algorithmic event we refer to in this section corresponds to an ASF event. (We treat several ASF invocations in response to a single MV event as a sequence of individual ASF events.) We begin with some tree-related definitions.

**Definition 6.8** (tree creation, active and permanent trees)**.** *A tree is said to be* created *at node $i$ at time $t$ if $i$ was not an active root at $t^-$, but is one at $t^+$. A tree is* active *if its root node is active. Otherwise, it is* inactive*. An active tree is* permanent *if it is never inactivated. A tree rooted at node $i$ is said to be* destroyed *at time $t$ if either $i$ ceases to be a root or a new tree is created in $i$ (i.e., $i$ becomes active after being inactive at $t^-$).*

Hereafter, we denote by $Tree_i$ a specific instance of a tree rooted at $i$. We refer to the time since this instance was created in $i$ until it was destroyed as the lifespan of $Tree_i$. Tree characteristics are specified by state tuples, relations, and predicates, defined next.

**Definition 6.9** (state tuple)**.** *The* state tuple *of a node $i$, $\chi_i$, is the group of variables that are exposed to other nodes (see ASF). Similarly, $\lambda_i(\chi_j)$ denotes the state tuple of node $j$ as known to $i$, and $\chi_u$ denotes the state carried by an update message $u$.*

**Definition 6.10** (relation conformance)**.** *Let $\mathbb{R}$ be a relation between two state tuples. A node $j$* conforms to $\mathbb{R}$ *if for every neighbor $m$ of $j$, and for every in-flight message $u$ sent from $j$ to $m$:*

- $(\chi_j, \lambda_m(\chi_j)) \in \mathbb{R}$.
- $(\chi_j, \chi_u) \in \mathbb{R}$.
- $(\chi_u, \lambda_m(\chi_j)) \in \mathbb{R}$.
- *For every in-flight message $u'$ sent before $u$ to $m$,* $(\chi_u, \chi_{u'}) \in \mathbb{R}$.

**Definition 6.11** (predicate conformance)**.** *Let $\mathbb{P}$ be a boolean predicate over state-tuples. A node $j$* conforms to $\mathbb{P}$ *if for every neighbor $m$ of $j$, and for every in-flight message $u$ sent from $j$ to $m$:*

- $\chi_j \in \mathbb{P}$ *(i.e., $\mathbb{P}(\chi_j) = true$).*
- $\chi_u \in \mathbb{P}$.
- $\lambda_m(\chi_j) \in \mathbb{P}$.

A tree conforms to a relation $\mathbb{R}$ (predicate $\mathbb{P}$) if all its nodes conform to $\mathbb{R}$ ($\mathbb{P}$). We proceed with formal definitions of the tree properties.

**Uniformity** Uniformity captures the sign and ID characteristics of trees that either have not exhibited fusion of opposite-signed charges since their creation, or a long time has passed since they exhibited such a fusion. In these trees, all active nodes have the same sign, active nodes can only increase their IDs, and every active node's ID is equal to or less than the root's.

**Definition 6.12** (uniform tree)**.** *An ordered pair of state tuples $(\chi_i, \chi_j)$ satisfies the* uniform relation $\mathbb{R}_u$ *if:*

$$\{T_j \neq 0\} \Rightarrow \{ID_i \geq ID_j\}.$$

*Let $i$ be a root node, and denote by $\overline{T}_i$ $i$'s last sign (i.e., the last non-zero value of $T_i$). A state tuple $\chi_v$ satisfies the* uniform predicate $\mathbb{P}_u$ *of $Tree_i$ if: (1) $T_v \in \{0, \overline{T}_i\}$; and (2) if $T_v \neq 0$ then $ID_v \leq ID_i$. We say that $Tree_i$ is* uniform *if it conforms to both $\mathbb{R}_u$ and $\mathbb{P}_u$.*

**Stability** A stable tree is an active tree in which for every active node, all nodes downtree from it are guaranteed to be active. This, in turn, guarantees that in the absence of expansion waves, the path from every active node to the root is fixed:

**Definition 6.13** (stable tree)**.** *An ordered pair of state tuples $(\chi_i, \chi_j)$ satisfies the* stable relation $\mathbb{R}_s$ *if:*

$$\{T_j \neq 0\} \Rightarrow \{T_i \neq 0\}.$$

*Let $i$ be a root node. We say $Tree_i$ is* stable *if it is active and conforms to $\mathbb{R}_s$.*

**Lemma 6.14.** *In any stable tree, an active node cannot be inactivated before the root.*

*Proof.* Let $i$ be a root of a stable tree, and let $j \in Tree_i$ be an active node. Therefore, it holds that $\lambda_j(T_k) \neq 0$ for $k = P_j$. Since $k$ conforms to $\mathbb{R}_s$, it follows that $k$ is also active and that $T_u \neq 0$ for any in-flight message $u$ from $k$ to $j$. Consequently, $j$ cannot be inactivated before $k$ is. As the same reasoning holds for $k$ and any node downtree from it, we have the result. ■

**Stable ID** A stable ID is a concept that is related both to stability and uniformity. An ID $x$ is stable if for every node $i$ that considers its neighbor $j$ to be an active node with ID $x$, then $j$ belongs to a stable and uniform tree. This implies that $j$ is active and has an ID=$x$ so any message that $j$ sends also has these traits, ensuring that $i$'s information (regarding $j$) is stable.

**Definition 6.15** (stable ID)**.** *An ID $x$ is stable if for every two neighboring nodes $j$ and $k$, if $\lambda_j(T_k) \neq 0$ and $\lambda_j(TID_k) = x$ then $k$ belongs to a stable uniform tree.*

**Inverse Stability** Inverse stability implies certain constraints on inverse hops of nodes whose ID is equal to their root's. We distinguish between in-tree and cross-tree inverse hops:

**Definition 6.16** (in-tree/cross-tree inverse hops)**.** *Let $j$ be an active node for which $\overline{P}_j = k$.*

- *$k$ is a* cross-tree *inverse hop of $j$ if $\lambda_j(ID_k) > ID_j$ and $\lambda_j(T_k) = -T_j$.*
- *$k$ is an* in-tree *inverse hop of $j$ if $\lambda_j(ID_k) = ID_j$, $\lambda_j(P_k) = j$, and $\lambda_j(\overline{W}_k) < \infty$.*

In inverse stable trees, every such cross-tree inverse hop must point to a node belonging to an opposite signed tree, and every such in-tree inverse hop must point to a node of the same tree that also has an inverse-hop of its own. Thus, such inverse hops form a path to an opposite-signed tree:

**Definition 6.17** (inverse stable tree)**.** *Node $k = \overline{P}_j$ is a* stable inverse hop *of node $j$ if $j$'s conditions on $\lambda_j(\chi_k)$ for $k$ being a valid inverse hop also hold for $\chi_k$ and any in-flight update message from $k$ to $j$. An ordered pair of state tuples $(\chi_i, \chi_j)$ satisfies the* inverse stable relation $\mathbb{R}_{is}$ *if:*

$$\{ID_i = ID_j\} \Rightarrow \{\overline{W}_j < \infty \Rightarrow \overline{W}_i < \infty\}.$$

*Let $i$ be a root. $Tree_i$ is* inverse stable *if for every node $j \in Tree_i$ such that $ID_j = ID_i$: (1) every inverse hop of $j$ is stable; (2) $j$ conforms to $\mathbb{R}_{is}$.*

**Full Inactivity** Finally, we distinguish between an inactive tree (whose root is inactive) and a fully inactive tree whose nodes are inactive:

**Definition 6.18** (fully inactive tree)**.** *A state tuple $\chi_v$ satisfies the* inactive predicate $\mathbb{P}_i$ *if $T_v = 0$. Let $i$ be a root node. We say that $Tree_i$ is* fully inactive *if it conforms to $\mathbb{P}_i$.*

In order to track tree characteristics over time, we make the following definitions that take into account the distances between nodes, messages and roots.

**Definition 6.19.** *Let $j$ be a node and $u$ an in-flight message sent by it. The distance between $j$ and $u$, denoted by $d(j,u)$, is the elapsed time since $u$ was sent.*

**Definition 6.20.** *A node $j$ conforms to a relation $\mathbb{R}$ up to height $H$ if the following conditions are met:*

1. *$\mathbb{R}$ includes the state-tuple pairs listed in definition 6.10 for all messages and neighbors whose distance from $j$ is less than or equal to $H$.*
2. *Let $m$ be a neighbor of $j$ such that $d(j,m) > H$. If there is no message $u$ from $j$ to $m$ such that $d(j,u) = H$, then one of the following conditions must hold:*
   - *There exists at least one in-flight message $u$ from $j$ to $m$ such that $d(j,u) > H$. Assume that $u$ is the most recent such message. It holds that $(\chi_j, \chi_u) \in \mathbb{R}$ as well as $(\chi_{u'}, \chi_u) \in \mathbb{R}$ for any in-flight message $u'$ sent after $u$.*
   - *There is no in-flight message $u$ from $j$ to $m$ such that $d(j,u) > H$. It holds that $(\chi_j, \lambda_m(\chi_j)) \in \mathbb{R}$ as well as $(\chi_{u'}, \lambda_m(\chi_j)) \in \mathbb{R}$ for every in-flight message $u'$ sent from $j$ to $m$.*

*Conformance to a predicate $\mathbb{P}$ up to height $H$ is defined similarly. Let $i$ be a tree root. $Tree_i$ conforms to a relation $\mathbb{R}$ (predicate $\mathbb{P}$) up to height $H$ at time $t$ if every node $j \in Tree_i$ such that $Dep_j(t) \leq H$ conforms to $\mathbb{R}$ ($\mathbb{P}$) up to height $H - Dep_j(t)$.*

In Appendix B, we show that trees conform to the uniformity, stability, and full-inactivity properties up to heights that depend on time. The proofs are technical and rely on a careful induction on events. Specifically, we prove the following:

**Lemma 6.21.** *For all $t \geq t_1$, every tree is uniform up to a height of (at least) $H(t) = t - t_1$.*

**Lemma 6.22.** *For all $t \geq t_1$, every active tree is stable up to a height of (at least) $H(t) = t - t_1$.*

**Lemma 6.23.** *Let $i$ be a root of an inactive tree that was inactivated at some time $t'$. For every $t > t'$, $Tree_i$ is fully inactive up to a height of (at least) $H(t) = t - t'$.*

From these lemmas and the fact that the graph is finite, it immediately follows that any tree that existed at $t_1$ becomes uniform and any such active tree becomes stable within finite time.

**Corollary 6.24.** *Let $i$ be a root of a tree at time $t_1$.*

- *$Tree_i$ is destroyed or becomes uniform within finite time after $t_1$.*
- *If $i$ is active at $t_1$, then $Tree_i$ it will become stable within finite time after $t_1$ assuming $i$ remains active.*

In addition, we show that trees created after $t_1$ are always uniform and stable (as long as they are active).

**Lemma 6.25.** *Let $i$ be a root of a tree created at time $t' > t_1$. The following holds for every $t > t'$:*

1. *$Tree_i$ is uniform.*
2. *$Tree_i$ is stable as long as $i$ remains active.*

Finally, we identify a sufficient condition for establishing a stable ID based on $d_{max}$ (the maximum edge delay in the graph), and establish several terms for inverse stability. The proofs are deferred to Appendix B.

**Lemma 6.26.** *Let $x$ be a tree ID. If all active trees with an ID of $x$ or higher are permanent after some time $t' \geq t_1$:*

1. *All active nodes with an ID=$x$ must belong to uniform stable trees within finite time.*
2. *$x$ becomes a stable ID in at most $d_{max}$ time after (1).*

**Lemma 6.27.** *Let $x$ be an ID, and assume that every ID $y > x$ is stable after some time $t' > t_1$. Let $i$ be a root of a stable tree at $t > t'$. If the following conditions hold:*

- *$Tree_i$ was created after $t_1$ or has been stable for at least $3d_{max}$ time before $t$.*
- *$Tree_i$ is uniform.*
- *$Tree_i$ either was created with an ID=$x$ after $t'$, has increased its ID to $x$ after $t'$, or was already inverse stable with an ID=$x$ at $t'$.*

*then $Tree_i$ is inverse stable.*

## 6.3 Stability of active roots

We now prove by induction on decreasing IDs that the set of active roots must be fixed (there are no more tree creations nor inactivations) within finite time after $t_1$. The main induction step is established by the following lemma.

**Lemma 6.28.** *Let $x$ be an ID. If all active roots with an ID $y > x$ are permanent, then all active roots with an ID of $x$ will be permanent within finite time.*

*Proof.* According to Lemma 6.26, there exists a time $t' > t_1$ by which every $y > x$ is a stable ID. Following corollary 6.24, there exists a time $t'' > t_1$ by which any active tree that existed at $t_1$ is uniform and stable. Let $t''' = max(t', t'' + 3d_{max})$. As the number of existing active trees with ID=$x$ at $t'''$ is finite, there exists a time by which this set of trees is fixed, i.e., remaining active trees are permanent. Following Corollary 6.24 and Lemma 6.25, every active tree that was either created after $t'''$ with an ID=$x$ or increased its ID to $x$ after $t'''$, satisfies the conditions of Lemma 6.27 and thus is inverse stable. We complete the proof by showing that such trees are permanent by contradiction.

Let $i$ be the root of such a tree, and assume by contradiction that $i$ is inactivated at some time $t > t'''$. This can happen only if $i$ develops an inverse hop, and MV subsequently sends the charge and inactivates the tree. Let $j^1 = \overline{P}_i$ at $t^-$. If $j^1$ is an in-tree inverse hop of $i$, inverse stability ensures that $j^1$ is directly uptree from $i$, has an ID=$x$, and also has an inverse hop $j^2 = \overline{P}_{j^1}$. Denote by $j^n$ the first node along this inverse hop chain that has a cross-tree inverse hop, $k$.

For every $1 \leq i \leq n$, $j^i$ will neither be inactivated nor change its next hop before if receives a message with $T = 0$ from its own next hop. Also, $j^i$ will not cancel its inverse hop unless it is inactivated as before, or until it receives a message from $\overline{P}_{j^i}$ indicating that $\overline{P}_{j^i}$ no longer presents a valid inverse hop. However, there can be no such messages in flight at $t^-$ because $Tree_i$ is both stable and inverse stable.

Since $k$ is a cross-tree inverse hop of $j^n$, it must have a stable ID and belong to a permanent, stable and uniform tree. Therefore, $k$ remains a valid inverse hop of $j^n$ (as long as $j^n$ remains active) even after $Tree_i$ is inactivated. Similarly, every $j^i$ $(1 \leq i \leq n)$ continues to hold a valid inverse hop after $t$ until it is inactivated by a message sent from $j^{i-1}$.

As MV transfers the charge from $i$ before inactivating $Tree_i$, when the charge reaches $j^1$ it is still active and has a valid inverse hop. Similarly, the charge will be transferred to $j^2, ..., j^n, k$ with no interruption. Note that during the transfer, a node $j^i$ can change its inverse hop towards another (stable) inverse hop chain ending with a different cross-tree inverse hop $k'$. Nevertheless, the new chain must offer a path whose weight is strictly less then the previous one. Therefore, in this case we just take $k'$ instead of $k$.

Denote $k$'s root by $r$. Once the charge reaches $k$, it will be deterministically forwarded along next hops to $r$, because $Tree_r$ is a permanent active tree. Therefore, the charge will either be fused with the one in $r$ or with any charge along the way within finite time (because the finite weight of the path on which the charge is routed). This contradicts our assumption that the number of distinct charges is constant after $t_1$. ■

**Lemma 6.29.** *The set of active trees is fixed within finite time after* $t_1$.

*Proof.* We initially prove that within finite time after $t_1$, all active trees are permanent. Observe that if $x$ is the highest ID among remaining charges, there cannot be any active root with a higher ID, so all active trees with an ID of $y > x$ are trivially permanent. Since IDs are positive integers, repeatedly applying Lemma 6.28 on decreasing charge IDs produces the result.

Next, we claim that no new roots will be created within finite time, because an active root permanently holds a distinct charge and the number of charges in the system is finite. After this time, the set of active roots remains fixed since active roots are neither created nor destroyed. ■

## 6.4 Convergence

In Appendix B, we show that the fact that the set of active roots becomes fixed (in finite time) implies in a straightforward manner that:

**Lemma 6.30.** *After $t_1$ all charges are of the same sign.*

**Lemma 6.31.** *MV terminates within finite time after $t_1$.*

To show that the algorithm terminates with the correct result, we build upon the following technical lemma:

**Lemma 6.32.** *For any connected component $X$, if no transfer messages are underway then:*

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

*Proof.* The following equation holds for every node $i$ at all times:

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

(Initially, the equation holds trivially for all nodes; 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:

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

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 $\sum_{i \in X} \sum_{j \in N^i} C_i(j)$ evaluates to zero, yielding the expected result. ■

**Theorem 6.33.** *MV stops within finite time after all external events have ceased with the correct output in every node.*

*Proof.* Termination is guaranteed from Lemma 6.31. 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 6.32, $\sum_{i \in X} C_i \geq 0$. Since Lemma 6.30 guarantees that all remaining charges must have the same sign, it follows that $\forall i \in X : C_i \geq 0$. Therefore, all nodes in $X$ decide *true*, as there are no negative trees in the graph. The situation when the majority decision should be $false$ is shown similarly. ■

# 7 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 uniform charge distribution because: 1) further vote changes are likely to 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 $H$ on the height of every tree once the algorithm has converged. Following any fixed number of vote changes, we derive an upper bound on convergence time in terms of the value of $H$. (Local convergence following topology changes is proved similarly.) Namely, given a bulk of $K$ single vote changes that hits a converged system, we show that the system converges again in at most $C \cdot H$ time, where $C$ is a function of only $K$, $H$, and other algorithm constants, rather than system size. While this bound is far from tight– in fact, $C$ is doubly exponential in $K$ – it is independent of the system size. Therefore, it is suitable for our purpose of proving locality from a pure theoretical point of view.

Thus, the height of any tree in the converged system is also $O(H)$. While this may imply that tree height can increase over time, it does not happen in practice. In the next section, we use simulations to verify our conjecture empirically, and demonstrate the local characteristics of our algorithm for arbitrary vote changes.

Let $t_0$ be the time at witch a set of $K$ vote changes take place. We make the following assumptions at $t_0$:

- The system is converged at $t_0^-$. There are no isolated nodes or disconnected components. (In other words, we reason about each connected component separately.)
- At $t_0^-$, the height of every tree is bounded by $H$.
- At $t_0^-$, node weights do note deviate too much from their node's depth. More formally, there exists a small constant $K_1$ such that the weight of every node is no higher than $K_1$ times its depth[3].

[3]We assume that changes in the physical topology are much less frequent than vote changes, which govern the construction of trees; recall that the most recent edge weight is considered whenever a next hop pointer is changed.

- The maximum delay of any edge (not necessarily a tree edge), $d_{max}$, is no longer than $H$.
- For brevity, we assume that edge delays are symmetric: for any two nodes $i$ and $j$, $d(i,j) = d(j,i)$.
- The vote changes do not change the majority decision. Moreover, when the system converges again, there exists at least one charge with the majority sign.

**Remark.** To simplify the presentation, we define $t_0 = 0$, so that every $t > t_0$ can be treated as absolute time rather than specifying it relative to $t_0$. (For example, we can say: "If $A$ holds at $t$, then $B$ holds by $2t$".) Also, we use the terms $O(H)$ and $const \cdot H$ extensively to express the fact that the time of a certain operation does not depend on system size. Usually, we do not specify *const* explicitly. It can refer to different values at different places. The existence of a constant factor is what matters.

Our locality proof builds upon the correctness proof in section 6, and follows a similar structure. Specifically, we provide timing bounds on certain processes of the algorithm (e.g., establishing uniform trees, developing stable inverse hops) whose correctness has been proved in section 6. What distinguishes the proofs in this section is that we do not assume in advance that all charge fusions cease at some finite time. Rather, we allow arbitrary fusion of same-signed charges and bound the time between successive fusions of opposite-signed charges. Thus, we define $t_1$ only to be the time by which all expansion waves die down, and whenever we rely on a lemma from section 6, we assume that no fusions of opposite-signed charges occur after some time $t' > t_1$ and apply the lemma with respect to $t'$. Throughout this section, we use the following notation:

**Definition 7.1.** *$D_{max}(t)$ is the maximum depth of any node at time $t$. $D_{active}(t)$ is the maximum depth of any* active *node at time $t$.*

We begin by bounding the duration of expansion waves and the maximum node depth during this period. The proofs are technical and are deferred to Appendix C.

**Lemma 7.2.** *All expansion waves die down by $t_1 = K_1 H$.*

**Lemma 7.3.** *For all $t > t_0$, $D_{max}(t) \leq 3t + const \cdot H$. Specifically, $D_{max}(t_1) = const \cdot H$.*

Once all expansion waves have died down, trees do not change their existing topology (but may expand over dead-branch nodes) unless they are inactivated. Active trees do not take over each other's nodes, and the algorithm's operation can be characterized by local charge transfers between small neighboring trees. Nevertheless, care must be taken to ensure that areas of activity remain local: trees can expand into large portions of the graph by inactivating opposite-signed trees along the way and taking over their nodes; therefore, tree state changes such as inactivations, new IDs, and sign flips, must be propagated to the tree's nodes at a faster pace than its expansion rate.

For example, consider a converged environment $E$ of the graph. Assume that a vote change introduces a single minority charge that is negligible compared to the net majority charge in $E$. Nevertheless, even this single vote change can result in the creation of a tree bearing a minority sign. Let $i$ be this tree's root. Clearly, the minority charge will be fused shortly (either at $i$ or at a nearby root) and $i$ will either be inactivated or flip its sign. Assume that $i$ is inactivated. For the algorithm to maintain local operation, $Tree_i$'s nodes must also be inactivated before $Tree_i$ expands too far. This way, $Tree_i$'s nodes can rejoin remaining majority trees in close proximity, terminating the protocol activity in $E$.

At first sight, it seems that the expansion rate of $Tree_i$ into this environment will be limited by the algorithm's reactive operation without introducing intentional delays because after all expansion waves die down, each existing $Tree_j$ in $E$ must be inactivated before any of its nodes can join $Tree_i$. Indeed, normally $Tree_i$'s expansion into $Tree_j$ will be delayed by at least $Tree_j$'s diameter (accounting for the inactivation and acknowledgement phases), resulting in a slowdown of 1/2 through $Tree_j$'s nodes. Therefore, inactivity or sign information, which is propagated along tree edges without delays, quickly reaches all $Tree_i$'s nodes.

Unfortunately, there are still pathological cases in which $Tree_i$ could expand unboundedly into $E$ if its expansion rate is not restricted. For these cases, we show that ASF's delay mechanism, which we introduced in section 4, ensures that a tree's expansion rate is at most half the speed permitted by the network delay. Based on this result, we bound the depths of active nodes. The proofs are based on bounding the possible node weights that can be assigned during join events, and are deferred to Appendix C:

**Lemma 7.4.** *For all $t > t_1$, $D_{active}(t) \leq \frac{t}{2} + const \cdot H$.*

**Lemma 7.5.** *There exists a constant $C_1$ such that for all $t > t_1 + d_{max}$, every node $i$ such that $Dep_i(t) > \frac{t}{2} + C_1 \cdot H$ is fully inactive (i.e., $i$ conforms to $\mathbb{P}_i$).*

Using results from section 6, we now show that trees acquire the stability, uniformity, and full inactivity properties locally:

**Lemma 7.6.** *Assume that no fusion between opposite-signed charges occurs after some time $t' > t_1$. Then:*

- *Let $i$ be the root of an active tree at $t'$. Assuming that $i$ remains active, $Tree_i$ will be stable by time $2t' + const \cdot H$.*
- *Let $j$ be the root of an inactive tree at time $t'$. $Tree_j$ will be fully inactive by time $2t' + const \cdot H$.*
- *Let $k$ be the root of a tree at time $t'$. $Tree_k$ will be uniform by time $2t' + const \cdot H$.*

*Proof.* We begin with stability. Let $H(t) = t - t'$, and assume that $Tree_i$ remains active after $t'$. According to Lemma 6.22, for every $t > t'$, $Tree_i$ is stable up to a height of $H(t)$. Following Lemma 7.5, for every $t > t_1 + d_{max}$ and for every node $j \in Tree_i$ such that $Dep_j(t) > \frac{t}{2} + C_1 \cdot H$, it holds that $j$ is fully inactive. Therefore, by solving $(t - t') - d_{max} = \frac{t}{2} + C_1 \cdot H$, we obtain that by time $t'' = 2t' + 2 \cdot C_1 \cdot + 2d_{max} = 2t' + const \cdot H$:

- Any node $l \in Tree_i$ such that $Dep_l(t'') \leq H(t'') - d_{max}$ conforms to $\mathbb{R}_s$ due to Lemma 6.22.

- Any node $l \in Tree_i$ such that $Dep_l(t'') > H(t'') - d_{max}$ conforms to $\mathbb{R}_s$ because it is fully inactive according to Lemma 7.5. (Note that $t'' > t_1 + d_{max}$ so it is justifiable to apply Lemma 7.5 here.)

Hence, $Tree_i$ is stable. Following lemmas 6.21 and 6.23, we use exactly the same reasoning to show that by time $t''$, it holds that $Tree_j$ is fully inactive and $Tree_k$ is uniform. ■

Our next step is to bound the time until all minority charges are fused. The crux of our proofs is to divide the time since $t_1$ into phases (or intervals). Each phase begins by assuming the highest possible tree heights and active node depths (according to $D_{max}$ and $D_{active}$) while ignoring previous phases, and ends with some property that holds. While this approach results in an exponential growth in elapsed time, the number of phases depends only on $K$, $H$, and other algorithm constants, rather than on system size. Unless noted otherwise, we will refer hereafter to fusion between opposite-signed charges simply as 'fusion'. We first bound the time between successive fusion events as follows:

1. We establish a limit on the highest ID that can be assigned to a minority charge. Next, by induction on decreasing charge IDs starting from the highest minority charge ID, we show that following a relaxation period after the last fusion took place, any root inactivation deterministically leads to fusion (if no fusion has already occurred by then).

2. We then show that it does not take too long for an inactive node with a weight of $\infty$ to join a neighboring active node. This observation is generalized to inactive trees in a straightforward manner.

3. By combining steps 1 and 2 we prove that, in the absence of fusions and inactivations, all inactive nodes join permanent active trees quickly, establishing a fixed forest of active trees that spans the entire graph. Therefore, any minority charge in the graph is either routed and fused at a majority root, or causes a root inactivation, which guarantees fusion.

**Lemma 7.7.** *For all $t > t_0$, the highest ID assigned to any minority charge is smaller than $2K\lambda_d$.*

*Proof.* At $t_0$, there are no minority charges in the system. Hence, the absolute value of the net minority charge inflicted by $K$ vote changes is at most $K\lambda_d$. Since no new minority charges are introduced after $t_0$, this value cannot increase. Consequently, the maximum ID assigned to any single minority charge is at most $2(K\lambda_d - 1) + 1 < 2K\lambda_d$. ■

**Lemma 7.8.** *There exist constants $C_1, C_2, C_3, C_4$ such that the following holds at every $t' > t_1$:*

*If an active root is inactivated at some time $t'' > C_1 t' + C_2 H$, then at least one fusion between opposite-signed charges either has already occurred in the interval $(t', t'']$, or will occur by $t''' = C_3 t'' + C_4 H$.*

The proof is based on the following technical lemma:

**Lemma 7.9.** *Let $x$ be a charge ID and $\widetilde{t}' > t_1$ a time after which every root with an ID of $y > x$ is not inactivated by inverse hops. Then, there exist constants $\widetilde{C}_1, \widetilde{C}_2, \widetilde{C}_3, \widetilde{C}_4 \geq 1$ such that if an active root with an ID $x$ is inactivated at some time $\widetilde{t}'' > \widetilde{C}_1 \widetilde{t}' + \widetilde{C}_2 H$ then at least one fusion between opposite-signed charges either has already occurred in the interval $(\widetilde{t}', \widetilde{t}'']$, or will occur by $\widetilde{t}''' = \widetilde{C}_3 \widetilde{t}'' + \widetilde{C}_4 H$.*

*Proof.* Currently assume that no fusion occurs after $\widetilde{t}'$. According to lemmas 6.25 and 7.6, by time $t^{(1)} = 2\widetilde{t}' + const \cdot H$ all active trees are stable, and every tree is uniform. In addition, every tree with an ID of $y$ whose root was inactivated before $\widetilde{t}'$ will be fully inactive by $t^{(1)}$. As a result, every active node with an ID of $y$ is guaranteed to be part of a stable uniform tree by $t^{(1)}$ because active roots with $y$ IDs are permanent. Recalling that $d_{max} = const \cdot H$, it follows from Lemma 6.26(2) that $y$ is a stable ID at $t^{(2)} = t^{(1)} + const \cdot H$.

According to Lemma 6.27, any tree with an ID $x$ after $t^{(2)}$ that: (1) was created after $t^{(2)}$, (2) increased its ID to $x$ after $t^{(2)}$, or (3) was inverse stable at $t^{(2)}$, is inverse stable. (For cases (2) and (3), we must also assume that trees that were created before $\widetilde{t}'$ had been active for at least $3d_{max}$ time. This can be guaranteed by taking $t^{(2)} > t^{(1)} + 3d_{max}$.) If there exists a tree at $t^{(2)-}$ that had an ID=$x$ but was not inverse stable, unstable inverse hops would inactivate its root within $D_{active}(t^{(2)})$ time unless the root ID is increased. Therefore, after $t^{(3)} = t^{(2)} + D_{active}(t^{(2)})$, all active trees with ID=$x$ are inverse stable.

Let $\widetilde{C}_1$ and $\widetilde{C}_2$ be such that $t^{(3)} = \widetilde{C}_1 \widetilde{t}' + \widetilde{C}_2 H$ (we bound $D_{active}(t^{(2)})$ using Lemma 7.4), and let $i$ be an active root with an ID=$x$ that is inactivated at $\widetilde{t}'' > t^{(3)}$. If $i$ was inactivated by fusion at $\widetilde{t}''$, or fusion has occurred in the interval $(\widetilde{t}', \widetilde{t}'')$, then we are done. Otherwise, $i$ was inactivated by a stable inverse hop at $\widetilde{t}''$. Following the arguments of Lemma 6.28, the charge of any active root $i$ with an ID=$x$ that develops a stable inverse hop is deterministically fused at a neighboring root $k$ (or with a charge along the way), as long as $k$ remains active. In this case, fusion takes place by $\widetilde{t}''' = \widetilde{t}'' + 2D_{active}(\widetilde{t}'') + d_{max}$, accounting for the path of active nodes in both $Tree_i$ and $Tree_j$, and the edge connecting them. By bounding $D_{active}(t')$ according to Lemma 7.4, we obtain $\widetilde{t}''' = \widetilde{C}_3 t + \widetilde{C}_4 H$ for matching values of $\widetilde{C}_3$ and $\widetilde{C}_4$. Note that according to our assumption, $k$ can be inactivated before $\widetilde{t}'''$ only due to fusion. ■

*Proof.* (Lemma 7.8) The proof is established by induction on decreasing charge IDs as follows: let $x$ be a charge ID; we assume that the lemma holds for inactivations of roots with ID $y > x$, and claim that it also holds for inactivations of roots with ID $y \geq x$. For the induction base, let $x_{max}$ be the highest possible ID that can be assigned to a minority charge after $t_0$ according to Lemma 7.7. Since the system was converged before $t_0$, there cannot be minority ID values greater than $x_{max}$ in any state tuple of any form (node $TID$

fields, message $TID$, etc.). Consequently, the lemma holds trivially for roots with an ID of $y > x_{max}$ because they cannot develop inverse hops, and remain active until their charge is fused. In addition, the lemma also holds for $x_{max}$ directly from Lemma 7.9.

For the induction step, denote by $\widehat{C}_1, \widehat{C}_2, \widehat{C}_3, \widehat{C}_4$ the constants for which the lemma holds with respect to IDs larger than $x$. Let $\widehat{t'} = \widehat{C}_1 t' + \widehat{C}_2 H$ and assume that some root $i$ with an ID of $y \geq x$ was activated at time $t'' > \widetilde{C}_1 \widehat{t'} + \widetilde{C}_2 H = \widetilde{C}_1 \widehat{C}_1 t' + (\widetilde{C}_1 \widehat{C}_2 + \widetilde{C}_2) H$, where $\widetilde{C}_i$ denotes the constants of Lemma 7.9. We distinguish among the following cases:

Case I: $i$'s ID is larger than $x$. Since $t'' > \widehat{t'}$, we obtain from the induction hypothesis that either fusion has occurred in the interval $(t', t'']$, or will occur by $\widehat{C}_3 t'' + \widehat{C}_4 H$.

Case II: $i$ has an ID=$x$. As long as roots with IDs larger than $x$ are not inactivated after $\widehat{t'}$, it follows from Lemma 7.9 that either fusion has occurred in the interval $(\widehat{t'}, t''] \subset (t', t'']$, or fusion will occur by $\widetilde{C}_3 t'' + \widetilde{C}_4 H$. However, if such inactivation has occurred at $\widehat{t''} \in [\widehat{t'}, \widetilde{C}_3 t'' + \widetilde{C}_4 H]$, we obtain from the induction hypothesis that either fusion has occurred in the interval $(t', \widehat{t''}]$, or fusion will occur by:

$$\widehat{C}_3 \widehat{t''} + \widehat{C}_4 \leq \widehat{C}_3 (\widetilde{C}_3 t'' + \widetilde{C}_4 H) + \widehat{C}_4 H =$$

$$\widetilde{C}_3 \widehat{C}_3 t'' + (\widehat{C}_3 \widetilde{C}_4 + \widehat{C}_4) H \triangleq t'''.$$

By taking $C_1 = \widetilde{C}_1 \widehat{C}_1$, $C_2 = \widetilde{C}_1 \widehat{C}_2 + \widetilde{C}_2$, $C_3 = \widetilde{C}_3 \widehat{C}_3$ and $C_4 = \widehat{C}_3 \widetilde{C}_4 + \widehat{C}_4$, the claim holds in all cases. Since Lemma 7.7 ensures that $x_{max} \leq 2K\lambda_d$, we conclude that the lemma holds for any active root with constants that depend only on $K$ (albeit exponentially), $H$, and other algorithmic constants rather than the system size. ■

While we have ensured that trees do not expand too fast by providing an upper bound on their expansion rate, we must also ensure that trees expand fast enough, because opposite-signed trees can at times be separated by regions of inactive nodes that impede the algorithm's progress. In Appendix C, we show that:

**Lemma 7.10.** *For every $t > 2K_1 H$ the following holds. Let $i$ be an active node, and $j$ a neighboring inactive node with $W_j = \infty$. If $i$ remains active, then $j$ becomes active in at most $4d(i,j)$ time.*

We are now ready to establish a bound on the time between successive fusion events:

**Lemma 7.11.** *There exist constants $\overline{C}_1, \overline{C}_2$ such that, for every $t' > t_1$, if there are minority charges in the system at $t'$ then fusion between opposite-signed charges will occur by $\overline{C}_1 t' + \overline{C}_2 H$.*

*Proof.* According to Lemma 7.8, if a root is inactivated at time $t'' > C_1 t' + C_2 H$, a fusion either has already occurred in $(t', t'']$, or will occur by $t''' = C_3 t' + C_4 H$. Let $t^{(1)} = C_1 t' + C_2 H$. If fusion has occurred in $(t', t^{(1)}]$ we are done. Therefore, assume this is not the case. The following statements hold as long as there are no root inactivations or fusions after $t^{(1)}$.

Let $j$ be the root of the last tree that was inactivated before $t^{(1)}$. Following Lemma 6.23, $Tree_j$ will be fully inactive by $t^{(2)} = 2t^{(1)} + const \cdot H$. Since inactive trees cannot expand

and there are no more tree inactivations, the depth of any inactive node after $t^{(2)}$ is at most $D_{max}(t^{(2)})$. Therefore, all inactive trees will increase the weight of all their nodes to $\infty$ by $t^{(3)} = t^{(2)} + 2d_{max} + 2D_{max}(t^{(2)})$, accounting for receiving Acks from neighboring nodes (not in the tree), propagating the Acks from the leaves to the root, and finally increasing the weights from the root to the leaves.

At $t_0$ every node in the graph was active. For a single vote change, the diameter of the largest region containing inactive nodes at time $t$ is $2t$, following information propagation considerations alone. Therefore, for $K$ vote changes, this diameter is at most:

$$D_{inactive}(t) = 2tK + (K-1)d_{max}$$

(accounting for $K$ disjoint regions connected by $(K-1)$ edges). Lemma 7.10 ensures that after time $2K_1H$, every inactive node $j$ with a weight of $\infty$ that is adjacent to an active node $i$ that remains active, will become active in at most $4d(i,j)$ time. These conditions hold for any inactive node after $t^{(3)}$. Consequently, all inactive nodes become active by $t^{(4)} = t^{(3)} + 4D_{inactive}(t^{(3)})$, essentially fixing the topology of all trees in the graph. Denote by $\overline{D} = D_{active}(t^{(4)}) = D_{max}(t^{(4)})$ the maximum depth of any node in the final topology.

Finally, we consider the possible states of a remaining minority charge $C$ in the system at $t^{(4)}$:

1. $C$ resides in an active root. Therefore, at least one pair of neighboring trees with opposite-signed roots exist. In this case, one of these trees will start developing inverse hops. If all nodes in this tree (and all messages sent by them) have identical IDs, then inverse hops will reach the tree's root and inactivate it in $\overline{D}$ time.

   Therefore, by taking into account the following additional time: (1) $2d_{max}$ to ensure that roots do not increase their ID by receiving same-signed charges ($d_{max}$ to notify all the neighbors of a root of its sign, and another $d_{max}$ to receive any same-signed charges that might have been sent before such notifications were accepted), (2) $\overline{D}$ to ensure all nodes in a given tree have identical IDs, and (3) another $d_{max}$ time to ensure that a tree's final ID reaches neighboring trees, we conclude that some root is inactivated.

2. $C$ resides in a tree rooted at another minority charge $C'$. This case is similar to 1) with respect to $C'$.

3. $C$ resides in a tree rooted at a majority charge. Here, $C$ will be routed and fused in $\overline{D}$ time. If $C$ is being transferred to such a tree, it adds at most $d_{max}$ time.

Let $t^{(5)} = t^{(4)} + 2\overline{D} + 3d_{max}$. If fusion occurs in the interval $(t^{(1)}, t^{(5)}]$, we are done. Otherwise, a root inactivation must occur at $t \in (t^{(1)}, t^{(5)}]$. Consequently, Lemma 7.8 guarantees that fusion must take place by $C_3t + C_4H \leq C_3t^{(5)} + C_4H$. By repeatedly substituting $t^{(5)}$, and bounding $D_{max}(t)$ and $D_{active}(t)$ according to lemmas 7.3 and 7.4, we obtain $C_3t^{(5)} + C_4H \leq \overline{C}_1t' + \overline{C}_2H$, for suitable values of $\overline{C}_1$ and $\overline{C}_2$. ■

Bounding the time by which all minority charges are fused is achieved by simply applying Lemma 7.11 repeatedly until all minority charges are exhausted. The proof is deferred to Appendix C.

**Lemma 7.12.** *There are no minority charges in the system in const · H time.*

Finally, for the algorithm to converge, all nodes must be active, uniform, and with fixed weights. While we have shown that properties such as tree activity and uniformity are achieved in bounded time, we have not yet bounded the propagation rate of weight changes. To do so, we build upon the following technical lemma, which is proved in Appendix C:

**Lemma 7.13.** *Let $i$ be an active node. If $i$ remains active and does not change its weight after some time $t > 2t_1$, then every uptree neighbor $j$ is active and does not change its weight by time $t' = t + 2d(i, j)$.*

**Theorem 7.14.** *Following any $K$ vote changes, the system converges in $O(H)$ time.*

*Proof.* According to Lemma 7.12, all minority charges are fused in $t' = const \cdot H$ time. Following the same arguments of Lemma 7.11, there exist constants $C_1$ and $C_2$ such that by time $t'' = C_1 t' + C_2 H$ all nodes are active and uniform, the tree topology is fixed, and $\overline{D} = D_{max}(t'') = const \cdot H$ bounds the maximum depth of any node in the graph. ($t''$ corresponds to $t^{(4)}$ in Lemma 7.11.) Furthermore, no inverse hops exist because this would imply an additional fusion between opposite-signed charges, contradicting the fact that there are no more minority charges in the system. Hence, no additional charge transfers occur.

Let $i$ be a root at $t''$, and $j$ an uptree neighbor of $i$. Since roots do not change their weights and $t'' > 2t_1$ (it can be easily verified that $t^{(4)} > 2t_1$ in Lemma 7.11), according to Lemma 7.13 $j$ does change its weight after $t'' + 2d(i, j)$. By induction, all nodes in $Tree_i$ do not change their weights by $t''' = t'' + 2\overline{D}$. Consequently, no additional update messages are sent. Accounting for any in-flight messages at $t''$, we conclude that the algorithm converges by time $t''' + d_{max} = const \cdot H = O(H)$. ■

# 8 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 percentage 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 ratios, thereby checking the sensitivity of the results to the proximity of the vote to the decision threshold.

Two representative unweighted 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 Yes/No ratio, and we measured the time until various fractions (90%, 95%, 100%, etc.) of the nodes decided

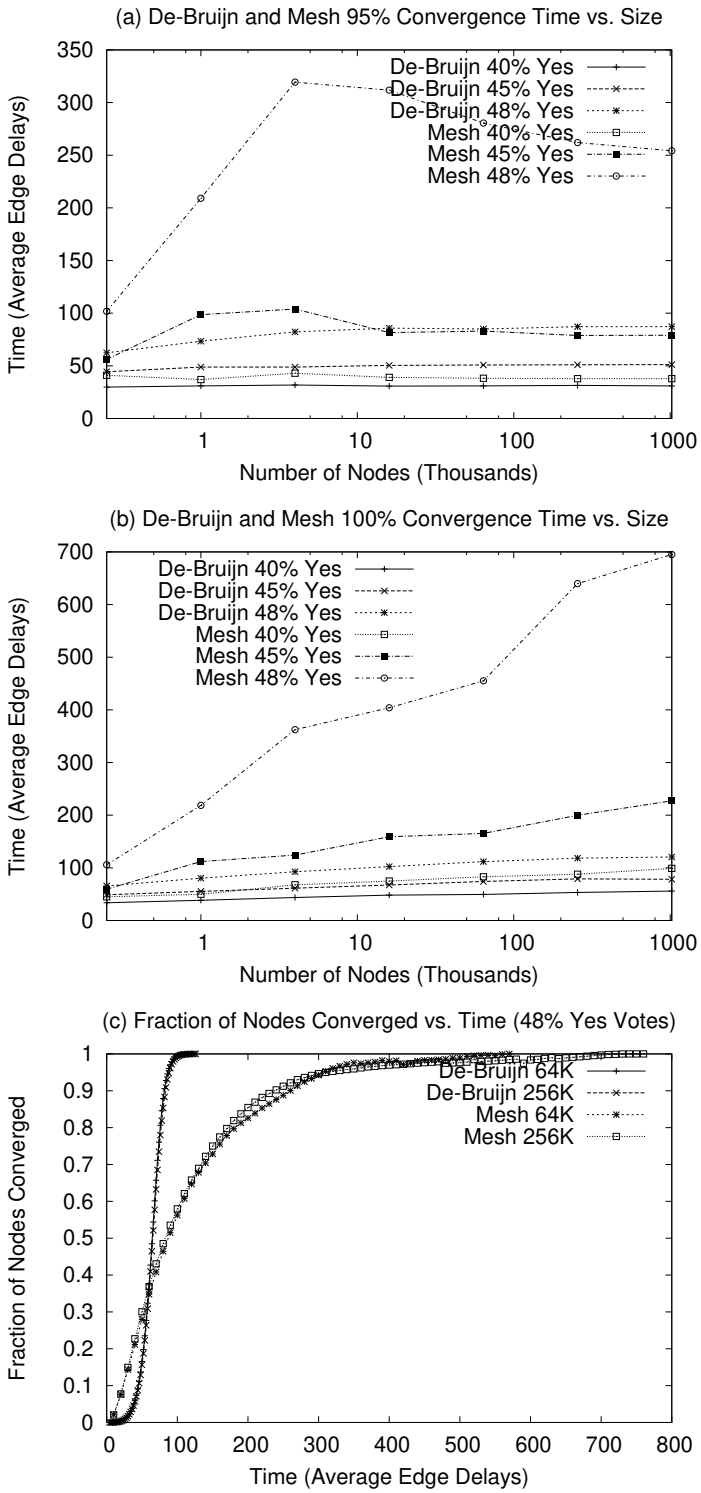


Figure 1: Bulk mode convergence and scale-up

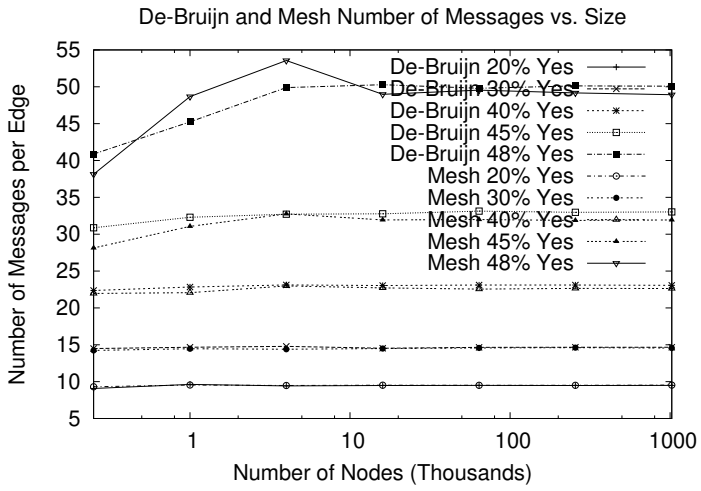


Figure 2: Messages per edge

on the correct outcome without subsequently retracting. Multiple experiments were carried out for each (graph type, size, Yes/No ratio) 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 convergence 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 Yes/No ratio and is essentially 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 as the Yes/No ratio approaches the threshold. Note that this worst case (over graph nodes) result is nonetheless averaged over multiple voting instances.

Figure 1 (c) focuses on the convergence percentile, providing the distribution 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 are barely distinguishable. This strongly suggests that locality and scalability hold for virtually every convergence 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 Yes/No ratio. As depicted in Figure 2, the number of messages per edge depends only on the Yes/No ratio 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 addition, all messages have a small constant size.

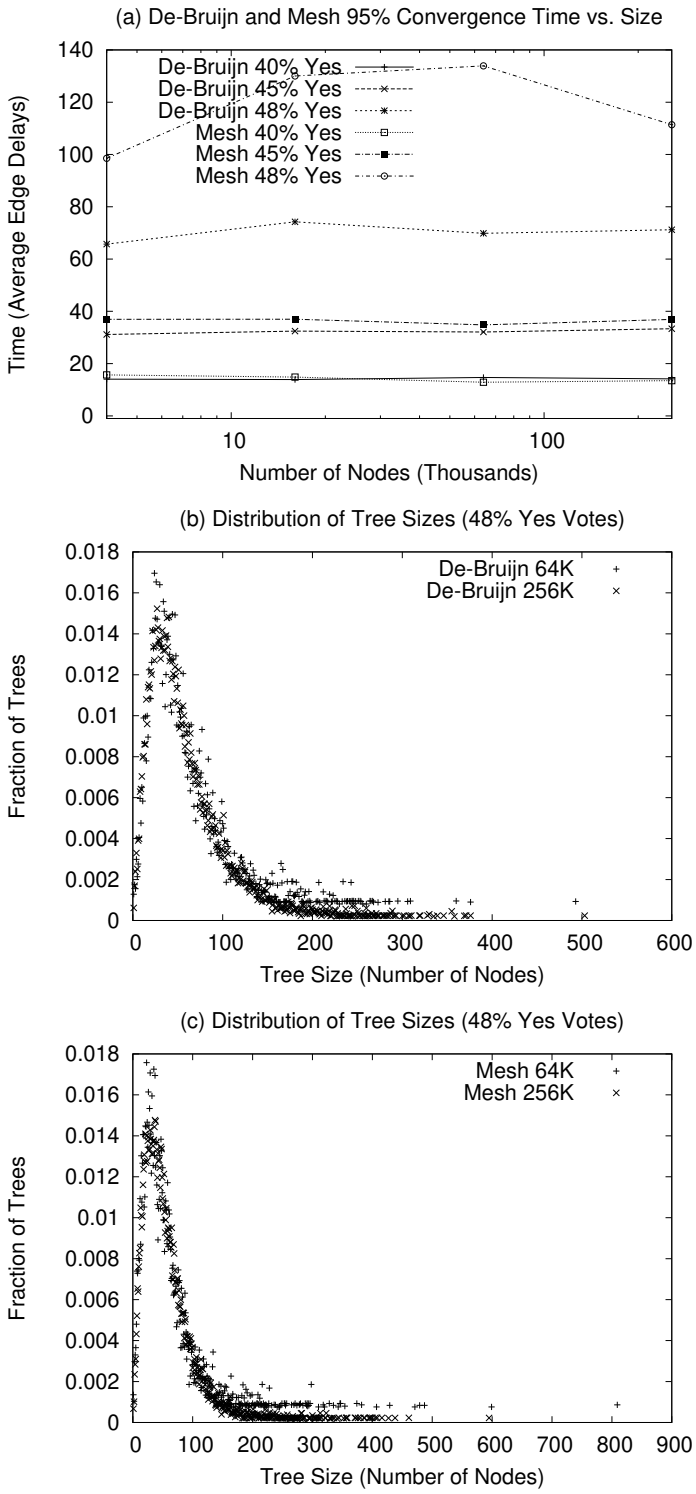


Figure 3: On-going mode convergence and locality

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 once every average edge delay. However, the overall Yes/No ratio remains constant. 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.

# 9 Related Work

Our work bears some resemblance to Directed Diffusion [10], a technique to collect aggregate information in sensor networks. As in [10], our routing is data-centric and based on local decisions. However, our induced routing tables are relatively short-lived, and do not require refreshment or enforcement. Our SF algorithm builds upon previous research in distributed Bellman-Ford routing algorithms that avoid loops such as [8] and [11].

Several alternative approaches such as sampling, pseudo-static computation and flooding can be used to conduct majority voting. With sampling the idea is to collect data from a small number of nodes selected with uniform probability, and compute the majority based on that sample. One such algorithm is the gossip based work of Kempe et al. [12]. Sampling, however, 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 every 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. [13] 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 that has been addressed by local algorithms is the problem of local mending or persistent bit. In this problem all nodes have a state bit that 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 [4], 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 [5]. 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, [6] also conducts majority votes in dynamic settings. However, their algorithm assumes that 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 [6] is faster in cases of a large majority, our algorithm is much faster as the majority is closer to the threshold.

# 10 Conclusions

We presented a local Majority Vote algorithm intended for dynamic, large-scale asynchronous systems. It uses an efficient, anytime spanning forest algorithm, which may also have other applications, as a subroutine. 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 on 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.

# References


[1] N. Linial, "Locality in distributed graph algorithms," *SIAM J. Computing*, vol. 21, pp. 193–201, 1992.

[2] D. Peleg, *Distributed Computing: A Locality-Sensitive Approach*. SIAM Monographs on Discrete Mathematics and Applications, 2000.

[3] B. Awerbuch, A. Bar-Noy, N. Linial, and D. Peleg, "Compact distributed data structures for adaptive network routing," *Proc. 21st ACM Symp. on Theory of Computing*, pp. 230–240, May 1989.

[4] S. Kutten and D. Peleg, "Fault-local distributed mending," *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, August 1995.

[5] S. Kutten and B. Patt-Shamir, "Time-adaptive self-stabilization," *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pp. 149–158, August 1997.

[6] 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.

[7] L. Ford and D. Fulkerson, *Flows in Networks*. Princton University Press, 1962.

[8] J. Jaffe and F. Moss, "A responsive routing algorithm for computer networks," *IEEE Transactions on Communications*, pp. 1758–1762, July 1982.

[9] F. Kaashoek and D. Karger, "Koorde: A simple degree-optimal distributed hash table," *In Proceedings of the Second Intl. Workshop on Peer-to-Peer Systems (IPTPS)*, February 2003.

[10] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: A scalable and robust communication paradigm for sensor networks," *In Proceedings of the Sixth Annual Intl. Conf. on Mobile Computing and Networking*, August 2000.

[11] J. Garcia-Luna-Aceves, "A distributed, loop-free, shortest-path routing algorithm," *Proceedings of IEEE INFOCOM*, pp. 1125–1137, June 1988.

[12] D. Kempe, A. Dobra, and J. Gehrke, "Computing aggregate information using gossip," *Proceedings of Fundamentals of Computer Science*, 2003.

[13] M. Bawa, H. Garcia-Molina, A. Gionis, and R. 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`.

# A SF Proofs

## A.1 SF Loop Freedom

*Proof.* (Lemma 3.3) Increases in $W_i$ are possible only when $T_i = 0$ (step 1), and therefore do not affect $\widehat{W}_i$ whose value is already $\infty$. So, the only increase in $\widehat{W}_i$ is due to $T_i$ becoming 0, setting $\widehat{W}_i$ to $\infty$. Because $i$ is nonisolated and $A_i$ is incremented in this case, $IsAck(i) = false$ (step 4). ■

*Proof.* (Lemma 3.4) 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 $j$, $\lambda_i(T_j)$ must be 1 and step 2 ensures that $W_i = \lambda_i(W_j) + d(i,j)$ and $T_i = 1$. Since $d(i,j) > 0$, we have $\lambda_i(\widehat{W}_{P_i}) < \widehat{W}_i < \infty$. Otherwise, $i$ remained with an existing next hop $j$. We are left with 3 possibilities: 1) if $\lambda_i(\widehat{W}_j)$ decreased, we return to the previous case; 2) if $\lambda_i(\widehat{W}_j)$ remained unchanged, so will $\widehat{W}_i$; 3) if $\lambda_i(\widehat{W}_j)$ increased, Lemma 3.3 guarantees that $\lambda_i(T_j) = 0$ and step 4 sets $T_i$ to 0, resulting in $\widehat{W}_i = \lambda_i(\widehat{W}_j) = \infty$. ■

*Proof.* (Lemma 3.5) If $T_i = 0$ since $i$ was initialized, then $i$ must trivially be a dead-branch node. Otherwise, let $t' < t$ be the last time $T_i$ changed from 1 to 0, and let $j$ be a neighbor of $i$. At $t'$, $A_i$ was increased and a message $u$ with $T = 0$ and $A_i$'s recent value was sent to every neighbor. After this time, $i$ did not send any messages with $T \neq 0$.

If the link between $i$ and $j$ was brought up between $t'$ and $t$, then $\lambda_j(T_i) = 0$ ever since. If the link between $i$ and $j$ was up during this time, the fact that $j$ has acknowledged $A_i$ ensures that there exists a time $t' < t'' < t$ in which $u$ was accepted by $j$ setting $\lambda_j(T_i) = 0$. Because $i$ did not send any messages with $T \neq 0$ after sending $u$, it follows from the FIFO ordering of messages that $\lambda_j(T_i)$ remains 0 at $t$.

Assume that $P_j(t) = i$. (Thus, the link between $i$ and $j$ was up at least since $t'$.) If $P_j$ did not point to $i$ at $t''^-$ it cannot point to $i$ at $t$, because SF never chooses an inactive neighbor (step 2). Similarly, $j$ could not have pointed to another node after $t''$ and subsequently returned to $i$ before $t$. Consequently, it holds that $P_j(t''^+) = P_j(t''^-) = i$, and $T_j(t''^+) = \lambda_j(T_i)(t''^+) = 0$. Since $j$ does not change its next hop, $T_j$ remains 0 and $IsAck(j)$ must evaluate to *true* before sending $i$ the Ack for $u$. Furthermore, $IsAck(j)$ must remain *true* at $t$ because $j$ is not inactivated again. Therefore, $j$ also satisfies the assumptions of the lemma at $t$. The proof is completed by repeatedly applying the same arguments for $j$ and any node uptree from it at $t$. ■

## A.2 SF Convergence

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. We initially show that all nodes remain acknowledged after some finite time $t_1$:

**Lemma A.1.** *There exists a time $t_1 > t_0$, s.t. for every $t > t_1$ $IsAck = true$ for all nodes.*

The proof builds on the following two observations:

**Lemma A.2.** *There exists a time $\widehat{t} > t_0$ s.t. for every $t > \widehat{t}$, if $T_j$ changes from 1 to 0 for some node $j$, then there exists a node $i$ for which $IsAck(i) = false$ and $W_i < W_j$.*

*Proof.* Let $\widehat{t}$ be a time by which all messages that were sent before $t_0$ have reached their destination or were dropped. For every $t > \widehat{t}$, $T_j$ changes to 0 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 $u$ it incremented $A_i$ and set $IsAck(i) = false$ (if it was not already $false$). $IsAck(i)$ will remain $false$ at least until $i$ receives from $j$ an Ack for $u$. Therefore, it follows from Lemma 3.6 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 $t^+$. ■

**Lemma A.3.** *If $IsAck(i) = false$ for some node $i$, $IsAck(i)$ will change to true within finite time.*

*Proof.* 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.* (Lemma A.1) We distinguish between two cases: (1) there are no active roots after $t_0$; and (2) there is at least one active root.

For (1), note that no new roots are created after $t_0$, so the number of these (inactive) roots becomes constant within finite time. According to Lemma A.3, a remaining root $i$ achieves $IsAck(i) = true$ within finite time. Because $i$ cannot become active (without ceasing to be a root), once $IsAck(i) = true$, it will remain so. Therefore, there exists a time $t_1$ by which all remaining roots have $IsAck = true$. Following Lemma 3.5, all nodes in the graph are dead-branch nodes, ensuring $IsAck = true$ for all nodes.

For (2), denote by $Wmin$ the minimum value of $W$ over nodes for which $IsAck = false$. Because $IsAck$ can change to $false$ only if $T$ changes to 0, it follows from lemmas A.2 and A.3 that $Wmin$ increases with time after $\widehat{t}$ (as defined in Lemma A.2). Moreoever, the increases are discrete. If there are no more nodes with $IsAck = false$ within finite time, we are done. Otherwise, denote by $Wmin(t), t > \widehat{t}$ the infinitely increasing monotone function of $Wmin$.

Let $U(t), t > \widehat{t}$ be the set of nodes $i$ for which $T_i(t) = 1$ and $W_i(t) \leq Wmin(t)$. $U$ is not empty, since it contains at least one active root, which is never unrooted.

A node $i$ that joins $U$ at some time $t' > \widehat{t}$ will never leave $U$. To see this, assume in contradiction that $i$ leaves $U$ at $t'' > t'$. Because $Wmin$ is monotone and $W_i$ cannot increase while $i \in U$ (weight increases are only possible if $T_i = 0$), $i$ can leave $U$ only if $T_i$ changes to 0. Therefore, according to Lemma A.2, there exists a node $j$ for which $IsAck(j) = false$ and $W_j(t'') < W_i(t'') \leq W_i(t') \leq Wmin(t') \leq Wmin(t'')$. This introduces a contradiction to $Wmin$'s minimality at $t''$.

We now claim that all nodes will be in $U$ within finite time. To see this, assume this is not the case. Therefore, there exists a time $t'$ such that $|U|$ is constant, and let $u \in U(t'), v \notin U(t')$ be neighboring nodes. Since $W_u$ cannot increase, and $Wmin$ increases forever, there exists a time $t''$ such that $W^* \triangleq \lambda_v(W_u) + d(u, v) < W(t'')$, and $\lambda_v(T_u) = 1$. Therefore, the following conditions must hold at $t > t''$:

- $W_v \leq W^*$ because otherwise, step 2 of SF would have ensured that $v$ joins $U$ by pointing to $u$. Therefore, $T_v$ must be 0.
- Given that $W_v \leq W^*$, $P_v$ cannot change because it would imply that $v \in U$. Similarly, $v$ cannot receive a message with $T = 1$ from $P_v$.
- Following Lemma A.3, $IsAck(v) = true$ within finite time, and will remain so. From this point onwards, $v$ adjusts its weight according to any update message from $P_v$ (step 1). Therefore, $v$ cannot receive an update message from $P_v$ with a weight higher than $W^* - d(v, P_v)$.

If $P_v \neq \perp$, then after some finite time the same conditions also apply to $P_v$ with an upper bound of $W^* - d(v, P_v)$ on its weight. Since the graph is loop-free and finite, after some finite time these conditions apply to a root node $r$. However, once $IsAck(r) = true$, $r$ increases its weight to $\infty$ (step 1) invalidating any upper bound on its weight, a contradiction.

Once all nodes are in $U$, no node will change to $IsAck = false$, guaranteeing that $IsAck = true$ for all nodes. ■

*Proof.* (Theorem 3.8) Let $t_1$ be the time stated in Lemma A.1. After this time, every inactive node $i$ adjusts its weight according to its next hop, or becomes active. Because active nodes remain active, inactive roots have a weight of $\infty$, and the graph is finite and loop-free, all inactive nodes will either join some active tree or increase their weight to $\infty$ at some time $t_2$. Therefore, if there are no active nodes at $t_1$, all nodes remain inactive and set their weight to $\infty$ at $t_2$, stopping the algorithm. If there is at least one active (root) node at $t_1$, all nodes are active by some time $t_2$.

As there are no weight increases ($T = 1$ for all nodes), a node can always choose the minimum path to a root among its current next hop and its other neighbors without restrictions. Hence, from this point onward the algorithm behaves exactly like the standard Bellman-Ford algorithm with two exceptions: (1) there are multiple destinations; and (2), some of the active nodes may not consider the latest edge weight to their next hop when calculating their own weight (recall that weight changes do not trigger an event in our algorithm). However, the algorithm's operation can still be simulated by a standard Bellman-Ford algorithm as follows.

Multiple roots can be simulated by zero-weighted edges connected to a single destination node, accounting for (1). As for (2), let $i$ be an active node such that $P_i = j$ and $W_i \neq \lambda_i(W_j) + d(i, j)$ after $t_2$. If $P_i$ changes, the algorithm will take into account the most recent value of $d(i, P_i)$. If $P_i$ does not change, consider the time by which $\lambda_i(W_j)$ is constant (such a time exists because $\lambda_i(W_j)$ is a non-increasing positive integer) and add a bogus event that sets $d(i, j)$ to $W_i - \lambda_i(W_j)$. (This event does not have any effect on the execution.) This operation can be applied to all such nodes within finite time. Hence, convergence follows from that of the standard Bellman-Ford algorithm. ■

## A.3 ASF Loop-freedom and Convergence

*Proof.* (Proposition 4.1) We show that the aforementioned adaptations do not compromise SF's correctness and convergence properties. 1-3) merely add information, and hence do not affect the behavior of the algorithm. 4) only restricts next-hop choices of active nodes, so it cannot cause loops. Nevertheless, 4) still enables inactive nodes to join active trees. 5) can imply that new roots can be created *after* the last *Root* operation was invoked. However, once such "pending roots" are created they cannot be unrooted, and there can only be a finite number of them. Therefore, there will be no root changes after some finite time, as in the original algorithm. Finally, 6) merely delays the normal operation of the algorithm, which does not affect any of our previous claims.

The proof of Lemma A.1 remains valid. Therefore, following the same line of arguments as in Theorem 3.8, either all nodes are active or all nodes are inactive within finite time (depending on the existence of an active root). If all nodes are inactive, there are no inverse hops, and all nodes set their weight to $\infty$ within finite time.

If all nodes are active, the topology is static (in ASF active nodes do not change their next hops) assuming that there are no expansion waves (we prove that all expansion waves die down within finite time following the cessation of all changes in Lemma 6.7). Hence, a node stabilizes its ID, activity state and weight once its direct downtree node is stable. Since roots are stable, a simple induction from the roots towards the leaves guarantees that these values are constant for all nodes within finite time. Similarly, a node stabilizes its inverse-hop pointer and weight once all its direct uptree nodes have stabilized theirs. A simple induction from the leaves toward the roots ensures that inverse-hop values are constant for all nodes within finite time. With no additional changes to the state of any node, the algorithm halts once the last update message is received. ■

# B MV Correctness Proofs

## B.1 Baseline Facts

*Proof.* (Lemma 6.6) An active root can be created at node $i$ between multiple $Root_i$ events and the first $UnRoot$ event (which deactivates the tree, if one was created). Following step 4 and the $IsRoot$ predicate of ASF, once such an active root is created, its ID ($TID_i$) equals $RID_i$ as long as $i$ remains active. As the only operation that changes $RID_i$ is $Root_i$, it takes an additional $Root_i$ operation (after the tree was created) to change $i$'s ID while it is still active. Therefore, two consecutive $Root_i$ operations with different IDs (without an intervening $UnRoot_i$ operation) must occur.

According to MV, different IDs represent different charges, and a $Root_i$ invocation always corresponds to the current charge at $i$. Therefore, when the first $Root_i$ operation is invoked, $i$ has non-zero charge. As a charge transfer triggers an $UnRoot_i$ operation (in step 2 of MV), there can be no such transfers between the $Root_i$ invocations. So, when the second $Root_i$ operation is invoked with a different ID, $i$ must have changed its charge without initially transferring it. Since there are no link failures or vote changes, this can happen only by fusion.

In order to decrease its ID, $i$'s charge would have to be fused with an opposite-signed charge. (Fusion of same-sign charges results in a larger residual charge in absolute value, and hence in a larger ID.) This contradicts our assumption that no such fusions occur. ■

*Proof.* (Lemma 6.7) Let $e_0$ be an expansion event that occurs before $t_0$, and let $\{e_n\}, n \geq 1$ be a chain of expansion events in $Close(e_0)$. Denote by $N_{e_n}$ the node at which $e_n$ occurs, and by $EL_{e_n}$ the value of $EL$ in any message sent during $e_n$. Since $EL_{e_n} = min(0, EL_{e_{n-1}} - d(N_{e_{n-1}}, N_{e_n}))$ and $t_{e_n} - t_{e_{n-1}} = d(N_{e_{n-1}}, N_{e_n})$, we conclude that the values of $EL_{e_n}$ decline to zero linearly with time, terminating the chain.

Every expansion event belongs to some chain of expansion events, and every such chain is part of an expansion wave, which originated in a vote change before $t_0$. Since expansion waves are created with finite $EL$ values, each of these chains terminates within finite time after $t_0$, stopping all expansion waves. ■

## B.2 Tree Properties

To prove the various tree properties stated in section 6.2, we make extensive use of the following special cases that guarantee that a node is or had been a dead branch:

**Lemma B.1.** *Let $i$ be a tree root, and let $t$ be $Tree_i$'s creation time. If $Tree_i$ was not created by a vote change, then $i$ had been a dead-branch node at $t^-$.*

*Proof.* According to the the $CanRoot$ predicate and step 4 of ASF, a new tree cannot be established in $i$ unless $T_i = 0$ and $IsAck(i) = true$. Following Lemma 3.5, $i$ must have been a dead-branch node at $t^-$. ■

**Lemma B.2.** *Let $i$ be a node that changes its next hop pointer at time $t$. If this change did not occur due to an expansion event, then $i$ had been a dead-branch node at $t^-$.*

*Proof.* According to the the $CanChange$ predicate and step 3 of ASF, $i$ can change its next hop pointer only if $T_i = 0$ and $IsAck(i) = true$. Following Lemma 3.5, $i$ must have been a dead-branch node at $t^-$. ■

Our proofs also relies on a property we call 'dead branch preserving'. After $t_1$, new trees can only be created at dead-branch nodes. This behavior limits the possible configurations of active/inactive nodes in trees. In order to describe these configurations, we introduce the notions of an extended path and state chains:

**Definition B.3** (extended path)**.** *Let $i$ be a root node. An* extended path $p$ of length $l$ *in $Tree_i$ is a directed route that starts at $i$, continues along tree edges (as defined by next hops but in the opposite direction) until it reaches a node $j \in Tree_i$, and ends in one of the following ways:*

1. *at $j$ and it holds that $Dep_j = l$.*
2. *at a neighbor $m$ of $j$ such that $P_m \neq j$ and $Dep_j + d(j, m) = l$.*
3. *at a point $q$ located on an edge incident to $j$, such that $Dep_j + d(j, q) = l$.*

*Note that in cases (2) and (3), the last edge may be traversed twice (in different directions).*

**Definition B.4** (state chain)**.** *Let $i$ be a root node. A* state chain $C$ *of length $l$ in $Tree_i$ is an ordered set of state-tuples along some extended path of length $l$ in $Tree_i$. $C$ includes the following state tuples:*

1. *For every two consecutive nodes $j$ and $k$ in the path ($j$ comes before $k$), $C$ includes $\chi_j$, $\lambda_k(\chi_j)$, and $\chi_u$ for every in-flight message $u$ from $j$ to $k$. If $P_k = j$ then $C$ also includes $\chi_k$.*

2. *If $j$ is the last node in the path, and the path ends at a point $q$ located on an edge incident to $j$, then $C$ includes $\chi_u$ for every message $u$ that $j$ sent on this edge that has not passed $q$.*

*The order of the state tuples is by increasing distances from the root, starting with $\chi_i$ whose distance is 0. For every other node $j \in Tree_i$ in the path, the distance of both $\lambda_j(\chi_{P_j})$ and $\chi_j$ is $Dep_j$. However, $\lambda_j(\chi_{P_j})$ is ordered before $\chi_j$. For every message $u$ in the path, the distance of $\chi_u$ is $Dep_j + d(j,u)$ where $j$ is the node that sent $u$. ($\chi_u$ is ordered after $\chi_j$ even if $u$ has just been sent.) Finally, if the path ends in a node $m \notin Tree_i$, then $\lambda_m(\chi_j)$ is the last state in the chain.*

As in the definitions of relations and predicates, we extend the previous definitions in order to track state chains over time:

**Definition B.5** (state chain extension)**.** *Let $i$ be a root, and let $C$ be a state chain of length $H$ in $Tree_i$. We define an* extension *for $C$ in the following cases:*

- *Assume that $C$ ends with the state of a message $u$, let $j$ be the node that sent $u$, and let $k$ be $u$'s destination. If $Dep_j + d(j,u) < H$, we define $C'$ as the extension of $C$ that includes the state of the latest in-flight message sent from $j$ to $k$ before $u$, or $\lambda_k(\chi_j)$ if no such message exists.*

- *Assume that $C$ ends with the state of a node $j \in Tree_i$. For every neighbor $k$ of $j$, we define $C'_k$ as an extension of $C$ that includes the state of the latest in-flight message sent from $j$ to $k$, or $\lambda_k(\chi_j)$ if no such message exists.*

A dead-branch preserving tree is simply a tree whose state chains conform to a limited set of patterns:

**Definition B.6** (dead-branch preserving state chains)**.** *Denote by "0" a state tuple for which $T = 0$, and by "1" otherwise. A state chain is* dead-branch preserving *if its state tuples can be described by one of the following regular expressions:*

- "$0^+$".
- "$1^+$".
- "$0^+1^+$".
- "$1^+0^+$".
- "$0^+1^+0^+$".

**Definition B.7** (dead-branch preserving trees)**.** *A tree is dead-branch preserving if all its state chains are dead-branch preserving. A tree is dead-branch preserving up to a height of $H$ if all its state chains of length $H$ as well as their extensions are dead-branch preserving.*

After $t_1$, every tree is dead-branch preserving up to a height that grows linearly with time:

**Lemma B.8.** *For all $t > t_1$, every tree is dead-branch preserving up to a height of $H(t) = t - t_1$.*

*Proof.* By induction on events. We first note that the validity of the lemma not only depends on the state of nodes and messages as determined by network events, but also on time. The requirements that a node's state has to conform to at a given time may change because of $H(t)$'s increase over time. More specifically, given a root $i$, a node $j \in Tree_i$, and a neighbor $m$ of $j$, any state chain of length $H(t)$ that includes $\lambda_m(\chi_j)$ must be dead-branch preserving starting from the moment that $H(t) = Dep_j + d(j, m)$. If $m$ is also uptree from $j$ at this time, the same holds for $\chi_m$. In order to reason about $Tree_i$'s state using a simple induction over discrete events, we define the notion of *implicit events*. An implicit event is an event that takes place at any such node $m$ for which $H(t) = Dep_j + d(j, m)$, given that no (real) event takes place in $m$ at this time.

Let $i$ be a root, $j$ a node uptree from $i$, and $m$ a neighbor of $j$. Consider an event at time $t > t_1$. We assume that the lemma was correct at $t^-$, and show that it must hold at $t^+$ by contradiction. For establishing the basis of the induction, we distinguish between two cases:

1. $Tree_i$ existed at $t_1$. At $t = t_1$, $Tree_i$ is trivially dead-branch preserving up to a height of 0: for every neighbor $k$ of $i$, the most recent in-flight message from $i$ to $k$ has the same state as $i$; if no such message exists, it holds that $\lambda_k(\chi_i) = \chi_i$; therefore, if $i$ sent an update message at $t_1$ then all zero-length state chains are either "00" or "11"; otherwise, the (single) zero-length chain in $Tree_i$ contains only $i$'s state, and any extension of this chain comprises two identical states as before.

2. $Tree_i$ is created at some time $t' > t_1$. According to Lemma B.1, $i$ was a dead-branch node at $t'^-$. Therefore, all state chains (of any length) at $t'^+$ are of the form "$110^+$". (Every chain starts with $i$'s active state followed by that of an active message sent by $i$ at $t'^+$, and ends with a sequence of inactive states.) Thus, $Tree_i$ is dead-branch preserving.

The lemma can apparently be violated in the following cases:

(a) Due to an event in $i$. If $i$ becomes an active root, all state chains of any length are dead-branch preserving as we have mentioned in the induction basis. If $i$ is inactivated (and sends a message to every neighbor), it replaces the initial "1" with "00" in every existing state chain. Thus, all state chains of length $H$ remain dead-branch preserving. If $i$ does not change its state the lemma holds trivially, a contradiction.

(b) $Dep_j(t) + d(j, m) < H(t)$, $P_m(t^-) = P_m(t^+)$ and the lemma is violated by $m$ receiving a message $u$ from $j$. When $u$ is received, $\lambda_m(\chi_j)$ is set equal to $\chi_u$. If $P_m \neq j$ or $\lambda_m(\chi_j)(t^-) = \chi_u$, no further changes occur. Otherwise, we distinguish between the following cases:

- $\lambda_m(\chi_j)$ changes from "1" to "0". If $\chi_m$ was not already "0", it changes to "0" as well and $m$ sends a message with "0" to every neighbor.
- $\lambda_m(\chi_j)$ changes from "0" to "1". Here, $\chi_m$ must have been "0" at $t^-$. At $t^+$, $\chi_m$ either remains "0" or it changes to "1" and $m$ sends a message with "1" to all its neighbors.

Let $C$ be a state chain of length $H(t)$ that includes $\lambda_m(\chi_j)$ and $\chi_m$. All the changes above can be seen as a sequence of state copying operations between neighboring states (adjusting $\lambda_m(\chi_j)$ according to $\chi_u$ or changing $\chi_m$ according to $\lambda_m(\chi_j)$), state duplication operations (sending a message according to $\chi_m$), or state elimination operations (dropping $\chi_u$ after adjusting $\lambda_m(\chi_j)$). These operations can only reduce one dead-branch preserving pattern to another (e.g, "$0^+1^+$" to "$0^+$"). Therefore, $C$ remains dead-branch preserving at $t^+$, a contradiction.

(c) $Dep_j(t)+d(j,m) < H(t)$, $P_m(t^-) = P_m(t^+)$ and the lemma is violated by a change in $\chi_m$ (without $m$ receiving a message at $t$). If $P_m \neq j$, the lemma holds trivially. Otherwise, the only possible change occurs when $\lambda_m(\chi_j)$ is "1" and $\chi_m$ is "0" at $t^-$, and $\chi_m$ changes to "1" at $t^+$ due to a clock tick event. This results in eliminating the previous value of $\chi_m$ and duplicating $\lambda_m(\chi_j)$ twice (by the new value of $\chi_m$ and an update message) in any state chain of length $H(t)$ that includes $\chi_m$. Thus, $Tree_i$ remains dead-branch preserving up to a height of $H(t)$.

(d) $Dep_j(t)+d(j,m) = H(t)$, $P_m(t^-) = P_m(t^+)$ and the lemma is violated by $m$ receiving a message $u$ from $j$. Denote by $C$ the state chain of length $H(t^-)$ at time $t^-$ that ends in $\chi_u$. According to the induction hypothesis, $C$ is dead-branch preserving. When $u$ is received, the state chain that ends in $\lambda_m(\chi_j)$ is identical to $C$ because $\lambda_m(\chi_j)$ is set to $\chi_u$, and $\chi_u$ is dropped. Therefore, if $P_m(t) \neq j$, the only state chain at $t$ of length $H(t)$ that includes $\lambda_m(\chi_j)$ is $C$, so the lemma holds.

If $P_m(t) = j$, ASF guarantees that if $\lambda_m(\chi_j)$ changed to "0" then so does $\chi_m$ (if it was not already "0"). Denote the state chain that ends with $\chi_m$ at $t^+$ by $C'$. Note that $C'$ is equal to the concatenation of $C$ and $\chi_m$ at $t^+$. Therefore, if $C$ ends with a "0", the concatenation of another "0" in $C'$ ensures that $C'$ is dead-branch preserving. If $C$ ends with a "1", it must match either the regular expression "$0^+1^+$" or "$1^+$". Consequently, concatenating it with either "0" or "1" (in $C'$) leaves it dead-branch preserving.

If $\chi_m$ changed then $m$ sends a corresponding message to every neighbor. In this case, all state chains of length $H(t^+)$ that include $\chi_m$ are derived from $C'$ by duplicating its last state. If $\chi_m$ has not changed, for every neighbor $k$ of $m$, either $\lambda_k(\chi_m) = \chi_m$ or $\chi_u = \chi_m$ for the most recent message $u$ sent from $m$ to $k$. Thus, any extension of $C'$ is also derived by duplicating the last state of $C'$. We conclude that all state chains of length $H(t)$ (or their possible extensions) at $t^+$ are dead-branch preserving, a contradiction.

(e) The lemma is violated by an implicit event in $m$. Since $m$ does not receive any message at $t$, there exists $\epsilon > 0$ such that: (1) no events occur in the system during the interval $[t-\epsilon, t]$; (2) $\epsilon < d(j,m)$; and (3) for every in-flight message $u$ from $j$ to $m$ at $t-\epsilon$, it holds that $d(j,u) < d(j,m) - \epsilon$.

Denote by $C$ the state chain of length $H(t-\epsilon)$ at time $t-\epsilon$ that includes all state tuples along the path from $i$ to $j$ and that of any in-flight message from $j$ to $m$. According to the induction hypothesis, it holds that both $C$ and its extension $C'$, which includes $\lambda_m(\chi_j)$, are dead-branch preserving at $t-\epsilon$. If $P_m(t) \neq j$, the only state chain at $t$ of length $H(t)$ that includes $\lambda_m(\chi_j)$ is identical to $C'$, so the lemma holds.

If $P_m(t) = j$, the only state chain at $t$ that includes $\chi_m$ is equal to the concatenation of $C'$ and $\chi_m$. Denote this chain by $C''$. According to ASF, it is not possible that $T_m \neq 0$ while $\lambda_m(T_j) = 0$. Therefore, if $C'$ ends with a "0", the concatenation of another "0" in $C''$ leaves it dead-branch preserving. If $C'$ ends with a "1", it must match either the regular expression "$0^+1^+$" or "$1^+$". Consequently, concatenating it with either "0" or "1" in $C''$ also leaves it dead-branch preserving. Finally, for every neighbor $k$ of $m$, either $\lambda_k(\chi_m) = \chi_m$ or $\chi_u = \chi_m$ for the most recent message $u$ sent from $m$ to $k$. Therefore, any extension of $C''$ merely duplicates the last state of $C''$, guaranteeing that it also is dead-branch preserving, a contradiction.

(f) $Dep_j(t) + d(j,m) = H(t)$, $P_m(t^-) = P_m(t^+)$ and the lemma is violated by a change in $\chi_m$ (without $m$ receiving a message at $t$). As we have shown in (e), the state chain that ends with $\lambda_m(\chi_j)$ at $t$, $C'$, is dead-branch preserving. Therefore, if $P_m \neq j$ then the lemma holds trivially. Otherwise, any state chain at $t^+$ of length $H(t)$ that includes $\chi_m$ is derived from $C'$ by duplicating its last state twice. (by the new value of $\chi_m$ and an update message). Consequently, $Tree_i$ is dead-branch preserving up to a height of $H(t)$, a contradiction.

(g) $Dep_j(t) + d(j,m) \leq H(t)$, and the lemma is violated by a change in $P_m$. We analyze this case with respect to state chains that include $\chi_j$. A change of $P_m$ from $j$ to another node only prunes some of the state chains that included $\chi_j$, thereby leaving them dead-branch preserving. Therefore, assume $P_m$ changed from some other node to $j$. If $Dep_j(t)+d(j,m) = H(t)$, this case similar to (d) or (f) because we do not need to take into account the states of nodes (or messages sent by them) that are uptree from $m$.

However, if $Dep_j(t) + d(j,m) < H(t)$, we must take these states into account. Denote the state chain that ends with $\lambda_m(\chi_j)$ at $t$ by $C$. If $m$ does not receive a message from $j$ at $t$, $C$ does not change at $t$ and hence is dead-branch preserving at $t^+$. If $m$ does receive such a message, $C$ must still be dead-branch preserving at $t^+$ following the same arguments as in (d). At $t^+$, any state chain of length $H(t)$ that includes $\lambda_m(\chi_j)$ is obtained by concatenating $C$, $\chi_m$, the state of a message $u$ sent by $m$ at $t^+$, and the states of nodes/messages/neighbors uptree from $m$. Since $\chi_u = \chi_m = \lambda_m(\chi_j)$ and Lemma B.2 ensures that $m$ was a dead-branch node at $t^-$, any such state chain is dead-branch preserving, a contradiction. ■

We rely on the propagation of the dead-branch preserving property to prove the tree properties of uniformity, stability, and full-inactivity:

**Uniformity** *Proof.* (Lemma 6.21) By induction on events. As in the proof of Lemma B.8, the validity of the lemma also depends on the flow of time. Therefore, we will make use of the notion of implicit events as defined there. Let $i$ be a root, $j$ a node uptree from $i$, and $m$ a neighbor of $j$. Consider an event at time $t$. We assume that the lemma was correct at

$t^-$, and show that it must hold at $t^+$ (assuming that $Tree_i$ still exists) by contradiction. As the basis for the induction, note that if $Tree_i$ existed at $t_1$, then it is trivially uniform up to a height of 0 at $t_1$. (We will address the case in which $Tree_i$ is created after $t_1$ shortly.) The lemma can be violated in the following cases:

(a) Due to an event in $i$. If $Tree_i$ is created at $t$, Lemma B.1 ensures that $i$ was a dead-branch node at $t^-$. Therefore, $Tree_i$ is trivially uniform (up to any height) at $t^+$. According to Lemma 6.6, as long as $i$ is an active root it cannot decrease its ID. In addition, $i$ cannot change its sign because there are no fusions of opposite-signed charges after $t_1$. Once $i$ is inactivated, it can obtain a new ID or sign only by creating a new tree or joining a different tree. Consequently, as long as $Tree_i$ exists, no event in $i$ can violate the conformance of any node in $Tree_i$ to either $\mathbb{R}_u$ or $\mathbb{P}_u$, a contradiction.

(b) $Dep_j(t)+d(j,m) \leq H(t)$, and the lemma is violated by a change in $\lambda_m(T_j)$ or $\lambda_m(ID_j)$. Such a change is possible only by receiving a message $u$ from $j$ at $t$. Therefore, it holds at $t^+$ that $\lambda_m(\chi_j) \in \mathbb{P}_u$, $(\chi_j, \lambda_m(\chi_j)) \in \mathbb{R}_u$, and $(\chi_{u'}, \lambda_m(\chi_j)) \in \mathbb{R}_u$ for any in-flight message $u'$ from $j$ because these assumptions held for $u$ at $t^-$, a contradiction.

(c) $m$ exhibits an implicit event and the lemma is violated because $\lambda_m(\chi_j)$ is not adjusted. Since $m$ does not receive a message at $t$, there exists $\epsilon > 0$ such that: (1) no events occur in the system during the interval $[t-\epsilon, t]$; (2) $\epsilon < d(j,m)$; and (3) for every in-flight message $u$ from $j$ to $m$ at $t-\epsilon$, it holds that $d(j,u) < d(j,m) - \epsilon$. According to the induction hypothesis, at $t-\epsilon$, $j$ conforms to $\mathbb{R}_u$ and $\mathbb{P}_u$ up to height $d(j,m)-\epsilon$. Since there is no in-flight message $u$ from $j$ to $m$ with $d(j,u) \geq d(j,m) - \epsilon$ at this time, it holds that $\lambda_m(\chi_j) \in \mathbb{P}_u$, $(\chi_j, \lambda_m(\chi_j)) \in \mathbb{R}_u$, and $(\chi_u, \lambda_m(\chi_j)) \in \mathbb{R}_u$ for any in-flight message $u$, by definition. Consequently, $j$ conforms to $\mathbb{R}_u$ and $\mathbb{P}_u$ at $t$, a contradiction.

(d) $Dep_j(t) \leq H(t)$, and the lemma is violated by a change in $T_j$ or $ID_j$, or by a message $u$ sent by $j$ due to such a change. Let $k = P_j(t^+)$. First, note that following ASF, if $T_j$ or $ID_j$ changed then $T_j = \lambda_j(T_{P_j})$ and $ID_j = \lambda_j(ID_{P_j})$. So, at $t^+$, it holds that $\chi_j \in \mathbb{P}_u$ because $\lambda_j(\chi_k) \in \mathbb{P}_u$ according to (b). Also, observe that if $j$ changes its next hop (and possibly its depth) at $t^+$, Lemma B.2 ensures that it must have been a dead-branch node at $t^-$. Since the lemma holds trivially in this case, we assume that $j$ did not change its next hop.

In the case of $Dep_j(t) = H(t)$, $j$ trivially conforms to $\mathbb{P}_u$ and $\mathbb{R}_u$ up to a height of 0 because it sends a message reflecting its new state to all its neighbors at $t^+$. Otherwise ($Dep_j(t) < H(t)$), $j$ must have conformed to $\mathbb{P}_u$ and $\mathbb{R}_u$ up to a height of $H(t) - Dep_j(t)$ at $t^-$. Consequently, it continues to conform to $\mathbb{P}_u$ up to this height at $t^+$. However, conformance to $\mathbb{R}_u$ is guaranteed only if $j$ does not decrease its ID. We next consider the possible cases in which $j$ can decrease its ID.

If $j$ is inactive at $t^+$, it follows from ASF that $j$ could have changed its ID only if both $IsAck(j) = true$ and $j$ was inactive at $t^-$. According to Lemma 3.5, in this case $j$ must have been a dead-branch node at $t^-$ so the lemma holds trivially.

Alternatively, if $j$ was inactive at $t^-$ and decreased its ID while being activated at $t^+$, we examine every possible state-chain $C$ of height $H(t)$ that includes $\chi_j$ at $t^-$. Since $\chi_j(t^-) =$ "0", either $\lambda_j(\chi_k)(t^-) =$ "1" or $j$ receives at $t$ a message $u$ from $k$ such that $\chi_u =$ "1". According to Lemma B.8, $C$ (and any extension of it) must follow the pattern "$1^+0^+$",

which ensures that $j$ was fully inactive (i.e., $j$ conforms to $\mathbb{P}_i$) at $t^-$ up to a height of $H(t) - Dep(j)$. Consequently, $j$ trivially conforms to $\mathbb{R}_u$ up to this height at $t^+$.

If $j$ was active at $t^-$ and remains active at $t^+$, then a decrease in $ID_j$ at $t$ implies that $\lambda_j(T_k) \neq 0$ and that $\lambda_j(ID_k)$ had also decreased. Since $\lambda_j(ID_k)$ can change only by receiving a message $u$ from $k$, and $k$ conforms to $\mathbb{R}_u$ at $t$, we reach a contradiction to the fact that $(\chi_u, \lambda_j(\chi_k)) \in \mathbb{R}_u$ at $t^-$.

(e) $Dep_j(t) = H(t)$, and the lemma is violated because $\chi_j$ is not adjusted following a change in $\lambda_j(\chi_k)$, where $k = P_j(t^-) = P_j(t^+)$. If $\lambda_j(T_k) = 0$ at $t^+$, then ASF ensures that $T_j(t^+) = 0$. Otherwise, ASF ensures that either $T_j(t^+) = 0$, or $T_j = \lambda_j(T_k)$ and $ID_j = \lambda_j(ID_k)$. Since $\lambda_j(\chi_k) \in \mathbb{P}_u$ according to (b), it holds that $\chi_j \in \mathbb{P}_u$ at $t^+$ in any case. Therefore, $\chi_j$ does not change at $t$ only if $\chi_j \in \mathbb{P}_u$ at $t^-$.

Furthermore, either $\lambda_m(\chi_j) = \chi_j$ or $\chi_u = \chi_j$ for the most recent message $u$ sent from $j$ to $m$. Consequently, $j$ conforms to $\mathbb{R}_u$ and $\mathbb{P}_u$ up to a height of 0 as required.

(f) $j$ exhibits an implicit event and the lemma is violated because $\chi_j$ is not adjusted. Let $k = P_j$. According to (c), it holds that $\lambda_j(\chi_k) \in \mathbb{P}_u$. Following the same arguments as in (e), either $T_j = 0$, or $T_j = \lambda_j(T_k)$ and $ID_j = \lambda_j(ID_k)$. Therefore, it holds that $\chi_j \in \mathbb{P}_u$, guaranteeing that $j$ conforms to $\mathbb{R}_u$ and $\mathbb{P}_u$ up to a height of 0.

(g) $j$ joins $Tree_i$ by changing its next hop pointer, such that $Dep_j(t) < H(t)$ and the lemma is violated indirectly by a node uptree from it. According to Lemma B.2, $j$ must be a dead-branch node at $t^-$, so the lemma holds trivially at $t^+$ with respect to any node uptree from $k$, a contradiction. ■

**Stability** *Proof.* (Lemma 6.22) Let $i$ be a root of an active tree at time $t$. The proof is immediate by observing that according to Lemma B.8, every state-chain of length $H(t)$ (and every extension of it) in $Tree_i$ must follow the pattern of either "$1^+$" or "$1^+0^+$" because $i$ is active. This guarantees that any node $j$ with depth $Dep_j \leq H(t)$ conforms to $\mathbb{R}_s$ up to a height of $H(t) - Dep_j$ by definition. ■

**Full Inactivity** *Proof.* (Lemma 6.23) By induction on events. Since the validity of the lemma also depends on the flow of time, we will use the notion of implicit events as defined in the proof of Lemma B.8. Let $j$ a node uptree from $i$, and $m$ a neighbor of $j$. Consider an event at time $t \geq t'$. We assume that the lemma was correct at $t^-$, and show that it must hold at $t^+$ by contradiction. The lemma can be violated in the following cases:

(a) Due to an event in $i$. When $Tree_i$ is inactivated, its sends a message with $T = 0$ to all its neighbors. Hence, $Tree_i$ conforms to $\mathbb{P}_i$ up to a height of 0 trivially. If $i$ becomes active (while remaining a root), it ceases to be an inactive tree by definition. Consequently, as long as $Tree_i$ remains inactive, every event in $i$ can only result in sending messages with $T = 0$, which leave $Tree_i$ fully inactive (up to a height of $H(t)$), a contradiction.

(b) $Dep_j(t) + d(j, m) \leq H(t)$, and the lemma is violated by a change in $\lambda_m(T_j)$. Such a change is possible only by receiving a message $u$ from $j$ at $t$. Therefore, it holds at $t^+$ that $\lambda_m(\chi_j) \in \mathbb{P}_i$ because these assumptions held for $u$ at $t^-$, a contradiction.

(c) $m$ exhibits an implicit event and the lemma is violated because $\lambda_m(\chi_j)$ is not adjusted. Since $m$ does not receive a message at $t$, there exists $\epsilon > 0$ such that: (1) no events occur in the system during the interval $[t-\epsilon, t]$; (2) $\epsilon < d(j,m)$; and (3) for every in-flight message $u$ from $j$ to $m$ at $t-\epsilon$, it holds that $d(j,u) < d(j,m) - \epsilon$. According to the induction hypothesis, at $t-\epsilon$, $j$ conforms to $\mathbb{P}_i$ up to height $d(j,m)-\epsilon$. Since there is no in-flight message $u$ from $j$ to $m$ with $d(j,u) \geq d(j,m)-\epsilon$ at this time, it holds that $\lambda_m(\chi_j) \in \mathbb{P}_i$ by definition, a contradiction.

(d) $Dep_j(t) \leq H(t)$, and the lemma is violated by a change in $T_j$ or by a message $u$ sent by $j$ due to such a change. Let $k = P_j(t^+)$. According to (b), it holds that $\lambda_j(T_k) = 0$ at $t^+$. Therefore, $j$ could not have changed its next hop to $k$ at $t$ if $P_j$ pointed elsewhere before the event. Furthermore, if $Dep_j(t) < H(t)$ then $T_j$ could not have changed because according to the induction hypothesis it holds that $T_j = 0$ at $t^-$. As a result, a change in $T_j$ can only occur when $Dep_j = H(t)$ and $T_j \neq 0$ at $t^-$, and $j$ is inactivated at $t^+$. In this case, $j$ sends a message with $T = 0$ to all its neighbors. Hence, $j$ conforms to $\mathbb{P}_i$ up to a height of 0 as required, a contradiction.

(e) $Dep_j(t) = H(t)$, and the lemma is violated because $\chi_j$ is not adjusted (or $j$ does not send a message) following a change in $\lambda_j(\chi_k)$, where $k = P_j(t^-) = P_j(t^+)$. According to (b), $\lambda_j(T_k)$ must be 0 at $t^+$. Therefore, $\lambda_j(T_k)$ could only have changed from a non-zero value to 0. If $T_j \neq 0$ at $t^-$, ASF guarantees that it would change to 0 at $t^+$. Thus, $T_j$ must have already been 0 at $t^-$. In this case, either $\lambda_m(\chi_j) = \chi_j$ or $\chi_u = \chi_j$ for the most recent message $u$ sent from $j$ to $m$. Consequently, $j$ conforms to $\mathbb{P}_i$ up to a height of 0 as required, a contradiction.

(f) $j$ exhibits an implicit event and the lemma is violated because $\chi_j$ is not adjusted or $j$ does not send a message. Let $k = P_j$. According to (c), it holds that $\lambda_j(\chi_k) \in \mathbb{P}_i$. Following the same arguments as in (e), so is $\chi_j$, and $j$ conforms to $\mathbb{P}_i$ up to a height of 0 as required, a contradiction.

(g) $j$ joins $Tree_i$ by changing its next hop pointer, such that $Dep_j(t) < H(t)$ and the lemma is violated indirectly by a node uptree from it. Let $k = P_j$ at $t^+$. This case is impossible because according to the induction hypothesis $\lambda_j(T_k) = 0$, a contradiction. ■

Interestingly, exactly the same proofs for showing the propagation of uniformity and stability over time can be used to show that active trees created after $t_1$ are always uniform and stable:

*Proof.* (Lemma 6.25) Following Lemma B.1, $i$ had been a dead-branch node at $t'^-$. Therefore, $Tree_i$ is trivially dead-branch preserving, uniform and stable (up to a height of $\infty$) at $t'^+$. For all $t > t'$, the proof is exactly the same as that of Lemma 6.21 (for uniformity) and Lemma 6.22 (for stability) by taking $H(t) = \infty$. ■

**Stable ID** The proof of Lemma 6.26 is based on the following observation:

**Lemma B.9.** *In finite time after $t_1$, all trees are uniform.*

*Proof.* After $t_1$, every tree in the system had either existed at $t_1$, or had been created afterwards. According to Lemma 6.25, trees created after $t_1$ are uniform at every instant.

According to corollary 6.24, every tree that already existed at $t_1$ is guaranteed to be uniform within finite time. Therefore, there exists a time $t \geq t_1$ after which all trees in the system are uniform at every instant. ■

*Proof.* (Lemma 6.26) After $t'$, any tree with an ID of $x$ or higher lies in the following categories:

- Trees that have been active since $t'$. According to corollary 6.24, such trees become stable within finite time.
- Trees created after $t'$. Lemma 6.25 ensures that such trees are stable at all times.
- Trees that were inactive at $t'$. Following Lemma A.3, it holds that $IsAck = true$ within finite time for every such tree. Therefore, these trees cannot contain active nodes after this time according to Lemma 3.5.

Consequently, any such tree is either stable or fully inactive within finite time after $t'$.

In addition, Lemma B.9 ensures that all trees are uniform within finite time after $t_1$ (and hence after $t'$), guaranteeing that any active node with an ID=$x$ belongs to a tree whose ID is $x$ or higher. Thus, there exists a time $t''$ by which every active node with an ID=$x$ belongs to a uniform stable tree, proving (1).

As for (2), let $t''' = t'' + d_{max}$ and let $j$ and $k$ be two neighboring nodes such that $\lambda_j(T_k) \neq 0$ and $\lambda_j(TID_k) = x$ at some time $t > t'''$. Therefore, the last message $u$ that was received by $j$ from $k$ also carried these values. Denote by $t_u$ the time $u$ was sent. If $t_u \geq t''$ then $k$ was part of a uniform and stable tree ever since $u$ was sent.

Otherwise ($t_u < t''$), assume in contradiction that at some time after $t_u$, $k$ sent a message with $T \neq T_u$ or $TID < x$. Let $u'$ be the first message with such values. If $u'$ was sent before $t'$, it must have arrived before $t'''$. Since messages are delivered in FIFO order and $u'$ was sent after $u$, this contradicts the fact that $u$ was the last message received by $j$. If $u'$ was sent after $t''$, it follows that $k$ changed its activity status or decreased its ID after it was part of a uniform and stable tree. This presents a contradiction to either the uniformity or the stability of $k$'s tree. Therefore, at $t''$, it holds that $T_k \neq 0$ and $ID_k \geq x$, guaranteeing that $k$ belongs to a uniform and stable tree since then. ■

**Inverse Stability** Inverse stability holds for certain uniform and stable trees given that all IDs higher than theirs are stable. To show this, we rely on the uniformity and stability characteristics of new trees that we have proved above, and on the following two technical lemmas:

**Lemma B.10.** *Let $i$ be a root of an active tree created after $t_1$, and let $j$ be an active node uptree from $i$. For every neighbor $k$ of $j$, if $\lambda_j(T_K) \neq 0$ and $\lambda_j(P_k) = j$, then $k$ is an active node uptree from $j$.*

*Proof.* Let $t'$ be the last time $j$ joined $Tree_i$. $j$ could have joined $Tree_i$ in three ways: by being uptree from $i$ (or $j = i$) when $Tree_i$ was created, by changing its next hop, or indirectly due to some node downtree from $j$ changing its next hop. Therefore, according to lemmas B.1 and B.2 $j$ must be a dead branch in any case at $t'^{-}$.

Let $t''$ be the time $k$ sent an Ack for $j$'s last inactivation before $t'$. (If such a time does not exist, let $t''$ be the last time the link between $j$ and $k$ was raised before $t'$.) Following the same arguments as in the proof of Lemma 3.5, if $k$ is not uptree from $j$ at $t''^{-}$, it will not be uptree from $j$ at $t'$. If $k$ is uptree from $j$ at $t''^{-}$, it will either remain uptree from $j$ with $T_k = 0$ until $t'$ or change its next hop to another node. Therefore, it is neither possible that $T_k \neq 0 \wedge P_k = j$ at $t''^{+}$, nor could an update message with such values be sent in the interval $(t''^{+}, t')$. Moreover, if $T_k \neq 0 \wedge P_k = j$ held at $t''^{-}$, $k$ would have sent an update message to $j$ with different values at $t''^{+}$ before sending the Ack.

Because $j$ must receive $k$'s Ack before joining $Tree_i$, it follows that at $t'^{-}$ there are no messages in transit from $k$ to $j$ with $T_k \neq 0 \wedge P_k = j$, and either $\lambda_j(T_k) = 0$ or $\lambda_j(P_k) \neq j$. As a result, the only possibility for $j$ to obtain $\lambda_j(T_k) \neq 0 \wedge \lambda_j(P_k) = j$, is by receiving these values from $j$ in an update message sent after $t'$. This implies that $k$ became an active node uptree from $j$ after $j$ joined $Tree_i$. Following Lemma 6.25, $Tree_i$ is stable ever since it was created. Therefore, according to lemmas B.2 and 6.14 $k$ must still be an active node uptree from $j$. ■

**Lemma B.11.** *Let $i$ be a root of an active tree created before $t_1$, and let $j$ be an active node uptree from $i$. Assuming $Tree_i$ is stable since some time $t' \geq t_1$, it follows that for every $t > t' + 3d_{max}$ and every neighbor $k$ of $j$, if $\lambda_j(T_K) \neq 0$ and $\lambda_j(P_k) = j$ then $k$ is an active node uptree from $j$.*

*Proof.* We distinguish between the following cases:

Case I: $j$ was an active node in $Tree_i$ since $t' + 2d_{max}$. If $\lambda_j(T_K) \neq 0$ and $\lambda_j(P_k) = j$ at $t$, then $k$ had been an active node directly uptree from $j$ at some point during the interval $(t' + 2d_{max}, t)$. Therefore, $k$ must remain so at $t$ because $Tree_i$ is stable throughout this period.

Case II: $j$ became an active node in $Tree_i$ (and remained one) at $t'' > t' + 2d_{max}$. Here, we also distinguish between two cases:

- If $j$ last joined $Tree_i$ after $t'$, Lemma B.2 ensures that $j$ was a dead-branch node beforehand, whether it joined $Tree_i$ directly (by changing its next hop) or indirectly. Following similar arguments as in the proof of Lemma B.10, if $\lambda_j(T_K) \neq 0$ and $\lambda_j(P_k) = j$ at $t$ then $k$ must have been an active node directly uptree from $j$ after it joined $Tree_i$. As we pointed out in case I, this mandates that $j$ remains so at $t$.

- If $j$ last joined $Tree_i$ before $t'$, then $j$ must have been inactive for at least $2d_{max}$ time before $t''$. (Note that $j$ could not have been activated and inactivated again during this time because $Tree_i$ is stable.) Consequently, by $t' + d_{max}$ every neighbor of $j$ had received a message informing it that $j$ is inactive, so $k$ could not have been an active node directly uptree from $j$ after this time (at least until $t''$). In turn, either $\lambda_j(T_K) = 0$

or $\lambda_j(P_k) \neq j$ since then, or $j$ had acquired these values by receiving a message from $k$ by $t' + 2d_{max}$. Therefore, if $\lambda_j(T_k) \neq 0$ and $\lambda_j(P_k) = j$ after $t''$, then $k$ must have been an active node directly uptree from $j$ after it joined $Tree_i$ as before.

■

*Proof.* (Lemma 6.27) by induction on events. We concentrate on proving that $Tree_i$ conforms to $\mathbb{R}_{is}$; the stability of its inverse hops will be shown as a byproduct. Let $i$ be the root of such a tree. Consider an event at time $t > t'$. We assume that the lemma was correct at $t^-$, and show that it must hold at $t^+$ by contradiction. Let $j$ be a node in $Tree_i$ at $t^+$, and let $m$ be a neighbor of $j$. The lemma can be violated in the following cases:

(a) The lemma is violated upon tree creation. According to Lemma B.1, $i$ was a dead-branch node at $t^-$. Thus, all nodes uptree from $i$ are dead-branch nodes at $t^+$. Since a value of $\overline{W} < \infty$ is only possible in an active state tuple (i.e., a state tuple with $T \neq 0$), all nodes uptree from $i$ conform to $\mathbb{R}_{is}$ trivially. On the other hand, $i$ conforms to $\mathbb{R}_{is}$ at $t^+$ whether it has a (cross-tree) inverse hop or not, a contradiction.

(b) The lemma is violated due to an event in $i$ that increases $ID_i$ to $x$. Since $Tree_i$ is uniform, at $t^+$, $ID_i$ is strictly higher than any active state-tuple in $Tree_i$. Therefore, $i$ conforms to $\mathbb{R}_{is}$ trivially, while no other active node in $Tree_i$ has an ID of $x$. Because $Tree_i$ is stable, if a node $j \in Tree_i$ is inactive neither $\lambda_m(\chi_j)$ nor any in-flight message that $j$ has sent have an active state (and thus do not have $\overline{W} < \infty$). Therefore, inactive nodes conform to $\mathbb{R}_{is}$ regardless of their ID, a contradiction.

(c) The lemma is violated by a change in $\lambda_m(\chi_j)$. However, this change can only occur by $m$ receiving an update message $u$ from $j$ at $t^-$, contradicting our assumption on $u$.

(d) The lemma is violated by a change in $j$'s state or a message sent by $j$. If $j$ was not in $Tree_i$ at $t^-$ and joined it at $t$, the case is similar to (a) because Lemma B.2 ensures that $j$ was a dead-branch node at $t^-$. If $ID_j \neq x$ at $t^+$, the lemma holds trivially. Likewise, if $j$ increased its ID to $x$ then $j$ trivially conforms to $\mathbb{R}_{is}$ due to $Tree_i$'s uniformity. (At $t^+$, $j$'s ID is strictly higher than that of any in-flight message $u$ sent by $j$ with $T_u \neq 0$, and hence with $\overline{W}_u < \infty$. The same holds for $\lambda_m(\chi_j)$.) Therefore, assume $ID_j(t^-) = ID_j(t^+) = x$ and that $j$ was already in $Tree_i$ at $t^-$ (and remained so at $t^+$). We distinguish between the following cases:

Case I: $j$ has an inverse hop $k$ at $t^+$. Here, $j$ conforms to $\mathbb{R}_{is}$ at $t^+$ whether it had an inverse hop at $t^-$ or not, because it conformed to $\mathbb{R}_{is}$ at $t^-$.

Case II: $j$ does not have an inverse hop at $t^+$. If $j$ did not have an inverse hop at $t^-$, it conforms to $\mathbb{R}_{is}$ at $t^+$ because it did at $t^-$. If $j$ had an inverse hop $k$ at $t^-$, it can lose it in the following cases:

1. $T_j$ changed. However, this contradicts the fact that $Tree_i$ is uniform and stable.

2. $\lambda_j(\chi_k)$ has changed such that $k$ does not qualify for being a valid inverse hop of $j$. Such a change is possible only by receiving a matching message $u$ from $k$. If $k$ was a cross-tree inverse hop, it held at $t^-$ that $\lambda_j(T_k) = -T_j$ and that $\lambda_j(ID_k) > x$. According to our assumptions $\lambda_j(ID_k)$ must be a stable ID, so $k$ belongs to a (different) uniform

and stable tree. Hence, it follows from $k$'s conformance to $\mathbb{R}_u$ and $\mathbb{R}_s$ at $t^-$ that $T_k = T_u = \lambda_j(T_k)$ and $ID_k \geq ID_u \geq \lambda_j(ID_k)$. This proves that every cross-tree inverse hop in $Tree_i$ is stable. As $\lambda_j(T_k)$ cannot change and $\lambda_j(ID_k)$ can only increase, $k$ must remain a valid inverse hop at $t^+$.

If $k$ was an in-tree inverse hop, it follows from either Lemma B.10 or Lemma B.11 that $k$ is an active node directly uptree from $j$. Since $Tree_i$ is uniform and stable, it holds at $t^-$ that $T_k = T_u = T_j$. Furthermore, $ID_k = ID_u = x$ because $\lambda_j(ID_k) = ID_j$ ($k$ is an in-tree inverse hop) and $\lambda_j(ID_k) \leq ID_k = \lambda_k(ID_j) \leq ID_j$ (uniformity). Therefore, $k$ also conforms to $\mathbb{R}_{is}$ at $t^-$, ensuring that $\overline{W}_k = \overline{W}_u < \infty$. This proves that every in-tree inverse hop in $Tree_i$ is stable. As a result, $k$ must remain a valid inverse hop at $t^+$.

We conclude that $j$ cannot lose a valid inverse hop without increasing its ID, a contradiction.

(e) The lemma is violated indirectly due to a node $k$ changing its next hop towards $j$. According to Lemma B.2, $k$ must be a dead-branch node at $t^-$, so the lemma holds trivially at $t^+$ with respect to any node uptree from $k$, a contradiction. ■

## B.3 Convergence

To Prove Lemmas 6.30 and 6.31, we first show that a fixed set of active roots implies that ASF converges independently from MV. Denote by $t_2$ the time by which the set of active roots is fixed.

**Lemma B.12.** *ASF converges within finite time after $t_2$.*

*Proof.* After $t_2$, the set of active roots is fixed. Any *Root* and *UnRoot* operations MV might invoke afterwards do not create nor destroy active roots, so ASF's operation is unaffected by them. Following Proposition 4.1, ASF converges within finite time. ■

Denote by $t_3$ the time by which ASF converges.

**Lemma B.13.** *If there is a least one charge in the system at $t_1$, there must be at least one active tree at $t_3$.*

*Proof.* Let $c$ be a charge that existed in the system at $t_1$. Since no fusion can take place after $t_1$, $c$ must still exist at $t_3$. Assume in contradiction that there are no active trees at $t_3$. Therefore, there are no active roots at $t_2$, and all nodes are inactive and acknowledged at $t_3$.

According to step 4 and the *CanRoot* predicate of ASF, any charged node that is inactive and acknowledged creates an active tree. So, $c$ must be carried by an in-flight transfer message at $t_3$. However, once this message reaches its destination a new active root will be created, contradicting the fact that the set of active roots is fixed after $t_2$. ■

*Proof.* (Lemma 6.30) Assume in contradiction that this is not the case. If two charges with opposite signs are active roots at $t_2$, there must exist two neighboring trees with opposite

signs by $t_3$. Since opposite signs imply different IDs, one of the trees must have developed inverse hops down to its root. This presents a contradiction either to the algorithm operation, which ensures that a root is inactivated once it develops an inverse hop, or to the fact that no roots can be inactivated after $t_2$. Therefore, there are no two active trees with opposite signs at $t_3$.

However, it follows from Lemma B.13 that there is at least one active tree at $t_3$. Therefore, all nodes are active and belong to trees with an identical sign $S$ by $t_3$. If two charges with opposite signs exist after $t_1$, one of them has a sign of $-S$. This charge will be routed along next hops and fused in finite time after $t_3$ if it has not been fused before, contradicting the fact that no fusion can take place after $t_1$. ■

*Proof.* (Lemma 6.31) If there are no charges at $t_1$, the algorithm halts once ASF converges at $t_3$. Otherwise, Lemma 6.30 ensures that all remaining charges are of the same sign. Therefore, at $t_3$ the signs of both the converged trees and remaining charges are identical. As a result, there can be no charge transfers after $t_3$. Once all in-flight transfer messages reach their destination, the algorithm halts. ■

# C MV Locality Proofs

## C.1 Expansion Waves

Expansion waves may be introduced to the system by vote changes at $t_0$. The algorithm's behavior as long as expansion waves are present requires special treatment, because unlike common operation (when all expansion waves die down), both active and unacknowledged nodes can change their next hops.

*Proof.* (Lemma 7.2) All expansion waves are generated by vote-change events at $t_0$. These events set the $EL$ value of any message they send equal to their node weight at $t_0^-$, which is bounded by $K_1 H$. Following the same arguments of Lemma 6.7, the last expansion event must occur before $t = K_1 H$. ■

In order to bound the maximum node depth, we first bound paths of dead-branch nodes and non dead-branch nodes separately, beginning with dead-branch nodes.

**Lemma C.1.** *The height of any dead-branch node is at most $t$.*

*Proof.* By induction on events. At $t = 0$, there are no dead branch nodes, so the lemma holds trivially. (Even if a root is inactivated at as a result of a vote change, it still has to notify its neighbors before it can become a dead-branch node.) Let $i$ be a node that incurs an event at time $t$. We assume that the lemma holds at $t^-$ and prove that it still holds at $t^+$.

If $i$ changes its next hop pointer, it follows from the algorithm that $T_i(t^+) \neq 0$, so $i$ is not a dead branch node at $t^+$. Furthermore, in this case there cannot be dead branch nodes downtree from $i$ at $t^+$ by definition. Therefore, changing a next hop pointer does not alter the height of any dead-branch node.

Since the lemma cannot be violated by tree topology changes, we only need to validate the lemma with respect to the node that exhibits the event. If $T_i \neq 0$ at $t^-$, $i$ cannot be a dead branch at $t^+$ because even if it were inactivated, it still has to wait for matching acknowledgements from its neighbors. If $T_i = 0$ and $IsAck(i) = true$ at $t^-$, Lemma 3.5 guarantees that $i$ is a dead branch. Therefore, the lemma holds at $t^+$, since it held at $t^-$. If $T_i = 0$ and $IsAck(i) = false$ at $t^-$, $i$ can become a dead-branch node if $IsAck(i) = true$ at $t^+$. In this case, we must ensure that $Height_i(t^+) \leq t$.

We examine every neighbor $j$ that is directly uptree from $i$ at $t^+$. Following the same arguments of Lemma 3.5, $j$ must have been a dead branch node that is directly uptree from $i$ ever since it sent an *Ack* for $i$'s last inactivation. Denote this time by $t'$. Therefore, the lemma holds with respect to $j$ at $t'$, and being a dead-branch node $j$ could not have increased its height in the interval $[t', t]$. In addition, it holds that $t \geq t' + d(i, j)$ because it takes $d(i, j)$ time for $j$'s *Ack* to reach $i$. ($i$ must receive this *Ack* by $t$.) As a result, $i$'s height through $j$ is bounded by:

$$Height_j(t') + d(i, j) \leq t' + d(i, j) \leq t.$$

Since this bound holds for all $j$, we have the result. ■

To bound paths of non dead-branch nodes, we initially handle the special case of nodes that exhibit expansion events.

**Lemma C.2.** *For any expansion event $e$, $Dep_e = W_e = t_e$.*

*Proof.* By induction on expansion events. At $t = 0$, every expansion $e$ occurs during a *Root* operation, for which $Dep_e = W_e = 0$. For $t > 0$, an expansion event $e$ occurs upon receiving a message $u$ sent by a previous expansion event $e'$. Denote by $i$ and $j$ the nodes in which $e$ and $e'$ occur, respectively. Since it takes $d(i, j)$ time for $u$ to reach $i$ and assuming that the lemma holds for $e'$, we have:

$$W_e = W_{e'} + d(i, j) = t_{e'} + d(i, j) = t_e.$$

Because $T_u \neq 0$, neither $j$ nor any node downtree from it could be a dead-branch node during the interval $(t_{e'}, t_e)$. Following Lemma 3.5, this implies that for any node $k$ downtree from $j$ or $k = j$, either $T_k \neq 0$ or $IsAck(k) = false$. Hence, $k$ cannot change its next hop unless it exhibits an expansion event during this interval. However, such an expansion event is not possible: according to the induction hypothesis, $W_{e'} = t_{e'}$ while any expansion event during the interval must have a weight that is strictly higher than $W_{e'}$; since $W_k \leq W_u = W_{e'}$ (Lemma 3.6), an expansion event at $k$ would present a contradiction. Therefore, $Dep_j(t_e) = Dep_j(t_{e'})$ because $j$'s path to its root does not change from $t_{e'}$ to $t_e$. Consequently:

$$Dep_e = Dep_{e'} + d(i, j) = t_{e'} + d(i, j) = t_e.$$

■

Now, we are able to bound the depth of any non dead-branch node using the following technical lemma:

**Lemma C.3.** *For every non dead-branch node $i$:*

1. *$Dep_i(t) < H + t' + t$, where $t'$ is the last time $i$ ceased to be a dead-branch node. (If $i$ has not been a dead-branch node since $t = 0$, set $t' = 0$.)*

2. *For every node $j$ downtree from $i$, if $Dep_i(t) - Dep_j(t) > H + t'$, then $W_j(t) < t$.*

*Proof.* By induction on events. At $t = 0$, the depth of all nodes is bounded by $H$, so the lemma holds trivially. We assume the lemma holds at time $t^-$, and prove that is still holds at time $t^+$. Let $i$ be some non dead-branch node at $t^+$. The lemma can be violated (with respect to $i$) in the following cases:

- $i$ exhibits an expansion event. According to Lemma C.2, $Dep_i = W_i = t$, which satisfies 1). Following Lemma 3.6, this implies that all nodes downtree from $i$ have $W < t$, which satisfies 2), a contradiction.

- $i$ becomes a non dead-branch node due to a join event in $i$. Let $j = P_i(t^+)$. Therefore, $T_i(t^+) = \lambda_i(T_j)(t^+) \neq 0$. This implies that either $\lambda_i(T_j)(t^-) \neq 0$, or $i$ received a message with $T \neq 0$ from $j$ at $t^-$. In both cases, $j$ is a non dead-branch node at $t$. Denote by $\widetilde{t'}$ the last time $j$ ceased to be a dead-branch node. (If $j$ was a non dead-branch node since $t = 0$, we define $\widetilde{t'} = 0$.) Assume for now that $t - \widetilde{t'} \geq d(i, j)$. Since the lemma holds for $j$ at $t^+$ (an event in $i$ does not affect $j$ because $i$ is uptree from $j$), we have:

  $$Dep_i(t) = Dep_j(t) + d(i, j) \leq$$
  $$(H + \widetilde{t'} + t) + (t - \widetilde{t'}) = H + 2t.$$

  So 1) holds. Furthermore, if there exists a node $k$ downtree from $i$ such that $Dep_i(t) - Dep_k(t) > H + t$ (note that $t' = t$), then:

  $$H + t < Dep_j(t) + d(i, j) - Dep_k(t)$$
  $$\leq Dep_j(t) - Dep_k(t) + t - \widetilde{t'}$$
  $$\Rightarrow H + \widetilde{t'} < Dep_j(t) - Dep_k(t).$$

  Therefore, 2) holds according to the induction hypothesis with respect to $j$ because an event in $i$ does not influence $j$ or any node downtree from it, a contradiction.

  We now justify our assumption that $t - \widetilde{t'} \geq d(i, j)$ by considering the following two situations:

  a) $j$ was never a dead-branch node after $t = 0$ ($\widetilde{t'} = 0$). Here, we examine the time it took for $i$ to be dead-branch node in the first place. (Recall that $i$ is assumed to be a dead-branch node at $t^-$.) At $t = 0$, there are no messages in-flight and $\lambda_j(T_i) \neq 0$. For $i$ to be a dead-branch node at $t^-$, it must have updated $\lambda_j(T_i)$ to 0 and received an *Ack* from $j$ beforehand. Since each of these events required a message, we have $t = t - \widetilde{t'} \geq 2d(i, j)$.

b) $j$ last ceased to be a dead-branch node at $\widetilde{t'} > 0$. Since $j$ was a dead-branch node at $\widetilde{t'}^{-}$, it holds that $\lambda_i(T_j)(\widetilde{t'}) = 0$ and $T = 0$ for every in-flight message from $j$ to $i$ by definition. $i$ cannot change its pointer to $j$ unless $\lambda_i(T_j)(\widetilde{t'}) \neq 0$. Therefore, at least one message with $T \neq 0$ was sent from $j$ to $i$ after $\widetilde{t'}$, ensuring that $t - \widetilde{t'} \geq d(i, j)$.

- A node $k$ downtree from $i$ increases its weight. Since $k$ is not a dead-branch node (otherwise $i$ would also be a dead-branch node), it follows from 3.5 that either $k$ is active or $IsAck(k) = false$. According to ASF, $k$ cannot increase its weight in this case, a contradiction.

- A node $k$ downtree from $i$ changes its next hop. Since $k$ cannot be a dead-branch node, this can happen only due to an expansion event in $k$. Therefore, it follows from Lemma C.2 that $Dep_k(t^+) = W_k(t^+) = t$. We distinguish between the following cases:

  a) $Dep_i(t^-) - Dep_k(t^-) > H + t'$. According to the induction hypothesis, $W_k(t^-) < t$. Therefore, an expansion event in $k$ presents a contradiction to our observation that nodes downtree from $i$ cannot increase their weight.

  b) $Dep_i(t^-) - Dep_k(t^-) \leq H + t'$. Therefore:

$$Dep_i(t^+) = (Dep_i(t^-) - Dep_k(t^-)) + Dep_k(t^+) \leq$$

$$H + t' + t$$

  which satisfies 1). Lemma 3.6 ensures that for any node $j$ downtree from $k$, $W_j(t^+) \leq W_k(t^+)$ thus satisfying 2), a contradiction.

Note that the passage of time itself cannot invalidate the lemma. ■

Finally, we can combine the previous results to achieve a bound on the maximum node depth:

*Proof.* (Lemma 7.3) Let $P$ be a path connecting a leaf node $l$ to a root node $r$ via next-hop pointers. If $r$ is a dead-branch node, it follows from Lemma C.1 that $Height_r(t) \leq t$. If $l$ is a non dead-branch node, Lemma C.3 guarantees that $Dep_l(t) \leq H + 2t$. Otherwise, $P$ must contain at least two nodes $i$ and $j$ such that $i = P_j$, $j$ is a dead-branch node, and $i$ is not. (It is not possible for $i$ and $j$ to switch roles because every node uptree from a dead-branch node is also a dead-branch node by definition.) In this case,

$$Dep_l(t) \leq Dep_i(t) + d(i, j) + Height_j(t) \leq$$

$$H + 2t + d_{max} + t \leq 2H + 3t.$$

Since $r$ and $l$ was chosen arbitrarily, we conclude that $D_{max}(t) \leq 2H + 3t$ for all $t$. Following Lemma 7.2:

$$D_{max}(t_1) \leq 2H + 3K_1H = const \cdot H.$$

■

## C.2 Tree properties

The proofs of Lemmas 7.4 and 7.5 are based on the following two technical lemmas:

**Lemma C.4.** *Let $e$ be a join event and $\{e_n\}, n \geq 1$ a chain of join events in $Close(e)$. The following holds for all $e_n$:*

$$W_{e_n} \leq W_e + (t_{e_n} - t_e) - (D_{e_n} - D_e) \tag{1}$$

*If in addition $D_{e_n} \geq W_{e_n}$, then:*

$$W_{e_n} \leq \frac{W_e + D_e}{2} + \frac{t_{e_n} - t_e}{2} \tag{2}$$

*Proof.* By induction. The lemma holds trivially for $e$. We assume that it holds for $e_{n-1}$ and prove that it also holds for $e_n$. Let $u$ be the update message sent by $e_{n-1}$ for which $e_n$ joins $e_{n-1}$, and let $L$ denote the edge delay between nodes $i$ and $j$ in which $e_n$ and $e_{n-1}$ have occurred, respectively. $u$ is accepted at $i$ at time $t_{e_{n-1}} + L$, updating $\lambda_i(D_j)$ and resetting $\Delta_i(j)$ to zero. For every $t > t_{e_{n-1}} + L$, the clock tick operation of ASF ensures that:

$$\Delta_i(j) \leq t - (t_{e_{n-1}} + L).$$

Therefore, at $t_{e_n}$ we have:

$$D_{e_n} = \lambda_i(D_j) + \Delta_i(j) \leq D_{e_{n-1}} + t_{e_n} - (t_{e_{n-1}} + L) \tag{3}$$

By substituting (3) in (1) for $e_n$, and applying the induction hypothesis for $e_{n-1}$ we get at $t_{e_n}$:

$$\begin{gathered} W_e + (t_{e_n} - t_e) - (D_{e_n} - D_e) \geq \\ W_e + (t_{e_n} - t_e) - (D_{e_{n-1}} + t_{e_n} - (t_{e_{n-1}} + L) - D_e) = \\ W_e + (t_{e_{n-1}} - t_e) - (D_{e_{n-1}} - D_e) + L \geq \\ W_{e_{n-1}} + L = W_{e_n}. \end{gathered}$$

This completes the proof for (1). If in addition $D_{e_n} \geq W_{e_n}$, we have:

$$W_{e_n} \leq W_e + (t_{e_n} - t_e) - (W_{e_n} - D_e)$$

$$\Rightarrow W_{e_n} \leq \frac{W_e + D_e}{2} + \frac{t_{e_n} - t_e}{2}$$

■

**Lemma C.5.** *Let $i$ be an active node. For all $t \in [t_0, t_1]$, $W_i \leq const \cdot H$.*

*Proof.* An active node or a node that becomes active changes its weight during join events. Therefore, we distinguish between the following cases:

1. $i$ has not changed its weight since $t_0$. Therefore, according to our assumptions at $t_0$ it follows that $W_i(t) = W_i(t_0) \leq K_1 Dep_i(t_0) \leq K_1 H$.

2. $i$ last changed its weight due to a join event $e' \in Close(e)$, where $e$ is a creation of a new tree (after $t_0$). Follwing Lemma C.4, $W_{e'} \leq W_e + (t_{e'} - t_e) - (D_{e'} - D_e)$. Since $W_e = D_e = 0$ and $D_{e'} \geq 0$, we have: $W_{e'} \leq t_{e'} \leq t_1$. According to Lemma 7.2, $t_1 \leq K_1 H$.

3. $i$ last changed its weight due to a join event $e'$, in which it became directly uptree from a node $j$ that has not changed its weight since $t_0$. Following 1), we have: $W_{e'} = W_j + d(i,j) \leq K_1 H + d_{max} \leq (1 + K_1)H$.

4. $i$ last changed its weight due to a join event $e' \in Close(e)$, where $e$ is a join event as in case 3). Following lemmas C.4 and 7.2, and noticing that $t_{e'} - t_e \leq t_1$ and $D_{e'} - D_e \geq 0$, we have:
$$W_{e'} \leq W_e + (t_{e'} - t_e) - (D_{e'} - D_e) \leq$$
$$(1 + K_1)H + t_1 \leq (1 + 2K_1)H.$$

We conclude that for all $t \in [t_0, t_1]$, $W_i(t) \leq (1 + 2K_1)H = const \cdot H$. ■

*Proof.* (Lemma 7.4) Let $i$ be an active node. We distinguish between the the following cases:

1. $i$ was active at $t_1$ and remained so at $t$. After $t_1$, a node cannot change its next hop pointer unless it is inactive and acknowledged. Following Lemma 3.5, a node that is inactive and acknowledge must be a dead-branch node. Therefore, neither $i$ nor any node downtree from it could have changed its next hop pointer since $t_1$. Consequently, $Dep_i(t)$ equals $Dep_i(t_1)$, which according to Lemma 7.3 is less than $const \cdot H$.

2. $i$'s last join event $e'$ belongs to $Close(e)$, where $e$ is a tree creation event that occurred after $t_1$. Since $W_e = D_e = 0$ and $D_{e'} \geq W_{e'}$, it follows from Lemma C.4 that:
$$W_{e'} \leq \frac{t_{e'} - t_e}{2} \leq \frac{t}{2}.$$
Consequently, $Dep_i(t) = Dep_i(t_{e'}) = W_{e'} \leq \frac{t}{2}$ because join events take into account the latest edge weight information, and no node in the event chain from $e'$ to $e$ could have changed its next hop pointer (due to similar arguments as in case 1).

3. $i$'s last joined a node $j$ after $t_1$ based on a message $u$ that was sent before $t_1$. Denote by $t_u$ the time $u$ was sent. (If $t_u < t_0$, let $t_u = 0$.) At time $t_u$, it follows from Lemma 7.3 that $Dep_j \leq const \cdot H$. After this time, neither $j$ nor any node downtree from it could have changed their next hop pointer (due to similar arguments as in case 1), so $Dep_j(t) \leq const \cdot H$ as well. Therefore, $Dep_i(t) = Dep_j(t) + d(i,j) \leq Dep_j(t) + d_{max} = const \cdot H$.

4. $i$'s last join event $e'$ belongs to $Close(e)$, where $e$ is a join event as in case 3. Let $j$ be the node in which $e$ takes place. As mentioned in case 2, join events take into account the latest edge weight information and no node in the event chain from $e'$ to $e$ could have changed its next hop pointer up to $t$. Moreover, $Dep_j$ remains constant after $t_e$. Therefore, $Dep_i(t) = Dep_i(t_{e'}) = Dep_j(t_e) + W_{e'} - W_e$. By applying Lemma C.4 with respect to $W_{e'}$, and noticing the fact that $D_e \leq k_1 W_e$, we have:

$$Dep_i(t) \leq Dep_j(t_e) + \frac{W_e + D_e}{2} + \frac{t_{e'} - t_e}{2} - W_e =$$

$$Dep_j(t_e) + \frac{D_e - W_e}{2} + \frac{t_{e'} - t_e}{2} \leq$$

$$Dep_j(t_e) + \frac{k_1 - 1}{2} W_e + \frac{t}{2}.$$

According to case 3 and Lemma C.5, both $Dep_j(t_e)$ and $W_e$ are bounded by $const \cdot H$. Therefore, $Dep_i(t) = \frac{t}{2} + const \cdot H$.

■

*Proof.* (Lemma 7.5) Let $C_1$ be the constant promised by Lemma 7.4, and assume by contradiction that there exists a node $i$ such that $Dep_i(t) > \frac{t}{2} + C_1 \cdot H$ but $i$ does not conform to $\mathbb{P}_i$. We distinguish between two cases:

1. $i$ has not changed its depth since $t - d_{max}$. According to Lemma 7.4, $i$ must have been an inactive node since $t - d_{max}$ because $Dep_i > D_{active}$ during the interval $(t - d_{max}, t]$. Therefore, at $t - d_{max}$, it holds that the last message $i$ has sent to any of its neighbors has $T = 0$, and so does any message that $i$ sends during $(t - d_{max}, t]$. By time $t$, it holds for every neighbor $j$ of $i$ that $\lambda_j(T_i) = 0$. Consequently, $i$ conforms to $\mathbb{P}_i$ at $t$.

2. The last time $i$ changed its depth was $t' \in (t - d_{max}, t]$. According to ASF, a node that changes its next hop becomes active. Thus, $i$ must have changed its depth at $t'$ due to some node $j$ downtree from it because according to Lemma 7.4, $i$ is inactive at $t'$. Following Lemma B.2, as a node uptree from $j$, $i$ was a dead-branch node at $t'$. Therefore, $i$ remained a dead-branch node at $t$ because it remained inactive, ensuring that it conforms to $\mathbb{P}_i$ at $t$.

A contradiction. ■

## C.3 Fusion Duration and Convergence

The proof of Lemma 7.10 builds upon the following three technical lemmas:

**Lemma C.6.** *For every active node that decreases its weight at time $t$, $W_i(t) \leq t$.*

*Proof.* By induction on events. Let $i$ be an active node that exhibits an event at time $t$. We assume that the lemma holds at $t^-$ and prove that it still holds at $t^+$. Let $j = P_i(t^-)$. $i$ can decrease its weight in two cases:

Case I: The event is an expansion event. Therefore, Lemma C.2 ensures that $W_i(t^+) = t$.

Case II: At $t^-$, it holds that $\lambda_i(W_j) + d(i,j) < W_i$ and $\lambda_i(D_j) + \Delta_i(j) > \lambda_i(W_j) + d(i,j)$. Assume that the last time $i$ chose $j$ as its next hop occurred at $t' > t_0$. At $t'^+$, it holds that $\lambda_i(W_j) + d(i,j) = W_i$. Therefore, at least one update message $u$ from $j$ was received by $i$ in the interval $(t', t)$, such that $W_u < \lambda_i(W_j)$ at the time it was received. Denote by $t_u$ the last time such a message was sent. At $t_u^-$, $j$ was active and decreased its weight (active nodes cannot increase their weight), so according to our assumption $W_j(t_u+) \leq t_u$. Therefore,

$$W_i(t^+) = W_u + d(i,j) = W_j(t_u) + d(i,j) \leq$$

$$t_u + d(i,j) \leq t.$$

If $i$ was active and directly uptree from $j$ ever since $t_0$, it is possible that $\lambda_i(W_j)+d(i,j) > W_i$ at $t_0$. (This can happen if $d(i,j)$ had increased before the system converged at $t_0$, while $i$ was already uptree from $j$.) In this case, we apply exactly the same reasoning with $t' = t_0$ and $W_u < W_i - d(i,j)$. ■

**Lemma C.7.** *A node that exhibits an expansion event does not decrease its weight before it is inactivated.*

*Proof.* By contradiction. Assume that $i$ is the first node that exhibits an expansion event $e'$ and subsequently decreases its weight at some time $t > t_{e'}$, while still active. Since $e'$ cannot be a root event (a root already has the minimal weight of 0), $e'$ must have occurred due to the reception of a message $u$ (with a positive expansion lifetime) from a neighbor $j$. Note that $j$ must have also exhibited an expansion event $e$ when it sent $u$. $i$ can decrease its weight in two cases:

Case I: $i$ experiences an additional expansion event $e''$ at $t$. According to C.2, it follows that $W_{e''} > W_{e'}$. Thus, $i$ must have increased its weight prior to $e''$. However, this contradicts ASF's operation that does not permit an active node to increase its weight.

Case II: $i$ exhibits a join event $e''$ at $t$. Since $i$ could not have exhibited an expansion event after $t_{e'}$ (without being inactivated), $i$ has not changed its next hop pointer ever since. Consequently, $i$ must have received an additional message $\widetilde{u}$ at $\widetilde{t} \in (t_{e'}, t]$ from $j$, such that $W_{\widetilde{u}} < \lambda_i(W_j)$. However, this means that when $\widetilde{u}$ was sent, $j$ also had decreased its weight. Denote by $t_{\widetilde{u}} \in (t_e, t_{e'})$ the time $\widetilde{u}$ was sent. This contradicts the assumption that $i$ is the first node to violate the lemma, because $j$ must have remained active at least until $t_{\widetilde{u}}$ and $t_{\widetilde{u}} < t_{e'}$. ■

**Lemma C.8.** *An active node can decrease its weight at most once before it is inactivated.*

*Proof.* By contradiction. Let $i$ be the first node that decreases its weight more than once during an active period. Denote by $t'$ and $t''$ the first and second times $i$ decreased its weight during this period, respectively.

Let $j = P_i(t'^-)$, and denote by $\widetilde{t}$ the last time $i$ chose $j$ as its next hop before $t'^-$. (If $P_i = j$ since $t_0$, let $\widetilde{t} = t_0$.) Following Lemma C.7, $i$ could not have experienced an expansion event at $t'$. According to Lemma C.6, $W_i(t') \leq t'$. Since $i$ cannot increase its weight while being active, it follows from Lemma C.2 that $i$ cannot exhibit an expansion event at $t''$. Therefore, it holds that $P_i = j$ during the interval $[\widetilde{t}, t''^+]$ because an active node can change its next hop pointer only through expansion events.

For $i$ to decrease its weight at $t'$, it must have received a message $u'$ from $j$ in $(\widetilde{t}, t']$, such that $W_{u'} < \lambda_i(W_j)$ (at the time the message was received). Likewise, $i$ must have received a message $u''$ from $j$ in $(t', t'']$, such that $W_{u''} < \lambda_i(W_j)$. Denote by $t_{u'}$ and $t_{u''}$ the time $u'$ and $u''$ were sent, respectively. Therefore, $j$ must have also decreased its weight at $t_{u'}$ and $t_{u''}$, while remaining active. Since $t_{u''} < t''$, we reach a contradiction. ■

*Proof.* (Lemma 7.10) According to ASF, $j$ will become active by pointing to $i$ as soon as $\lambda_j(D_i) + \Delta_j(i) > \lambda_j(W_i) + d(i,j)$ (unless $j$ is activated earlier). Therefore, it is sufficient to show that this condition holds by $t + 2d(i,j)$. Note that all the condition terms are non-negative. We distinguish between two cases:

Case I: $D_i \geq W_i$. Assume for now that $i$ does not change its weight after $t$. If $j$ has already received $i$'s latest weight and activity state, then $\lambda_j(D_i) \geq \lambda_j(W_i)$ at $t$. Hence, $j$ will be activated by $t+d(i,j)$ because $\Delta_j(i)$ increases linearly with time until the condition holds. Otherwise, there must exist at $t^+$ an in-flight update message $u$ from $i$ to $j$, which contains $i$'s recent weight and activity state. $u$ will reach $j$ at most by $t + d(i,j)$, resetting $\Delta_j(i)$ to zero. Afterwards, $\Delta_j(i)$ increases linearly with time, guaranteeing that $j$ will become active by $t + 2d(i,j)$. According to Lemma C.8, $i$ can decrease its weight at most once after $t$. (Increases are not possible at all because $i$ is active). Therefore, if $i$ changes its weight after $t$, it can postpone $j$'s activation at most by $2d(i,j)$.

Case II: $D_i < W_i$. This case is possible only if $i$'s most recent weight change or activation has occurred due to an expansion event $e$. Since all expansion events die down by $K_1 H$ (Lemma 7.2) and $W_e = t_e$ (Lemma C.2), it follows that $t_e < K_1 H$ and $W_i$ is at most $K_1 H$. In addition, $W_i$ cannot decrease (Lemma C.7) nor increase ($i$ is active). Therefore, by time $K_1 H + d(i,j) > t_e + d(i,j)$, $j$ must have received from $i$ an update message reflecting $i$'s state at $t_e$, resetting $\Delta_j(i)$ to zero. Because there are no further updates to $\lambda_j(W_i)$, $\Delta_j(i)$ increases linearly with time until $\Delta_j(i) > \lambda_j(W_i) + d(i,j)$, which occurs at most by time $2K_1 H + 2d(i,j)$. Thus, the lemma holds for all $t > 2K_1 H$. ■

*Proof.* (Lemma 7.12) The absolute value of the net minority charge inflicted by $K$ vote changes is at most $K\lambda_d$. Therefore, at most $K\lambda_d$ fusion events between opposite-signed charges are required to eliminate all minority charges in the system.

If no such fusions occur before $t_1$, Lemma 7.11 guarantees that at least one fusion occurs by $t = \overline{C}_1 t_1 + \overline{C}_2 H$. Let $\overline{C} = max(\overline{C}_1, \overline{C}_2)$. Applying the lemma once more with respect to $t$ ensures that at least two (separate) fusions occur by $\overline{C}_1 t + \overline{C}_2 H = \overline{C}_1^2 t_1 + (\overline{C}_1 \overline{C}_2 + \overline{C}_2) H < \overline{C}^2 t_1 + 2\overline{C}^2 H$. Therefore, the last fusion must occur by $C^{(K\lambda_d)} t_1 + 2C^{(K\lambda_d)} H$. Since $t_1 = const \cdot H$, we have the result. ■

*Proof.* (Lemma 7.13) We distinguish between three cases:

Case I: $i$ did not experience a join event since the system was converged at $t_0^-$. In this case, it holds at all times that for every neighbor $j$ of $i$: $\lambda_j(D_i)+\Delta_j(i) = \alpha(\lambda_j(W_i)+d(i,j)) \geq \lambda_j(W_i) + d(i,j)$. Consequently, if $j$ is directly uptree from $i$ at $t'$, it must have adjusted its activity state and weight according to $i$'s most recent values due to step 3 of ASF. Moreover, after $t'$ $j$ remains active and maintains its weight because it does not receive new values from $i$ nor change its next hop pointer. (Active nodes can change their next hop only due to expansion events, which cease by $t_1 < t'$. )

Case II: $i$ exhibited a join event at $t_0$ or afterwards, and $D_i(t) < W_i(t)$. In this case, $i$'s last weight change must have been due to an expansion event $e$. Since $t_e \leq t_1$, it follows from Lemma C.2 that $W_i(t_e^+) < t_1$. At $t_e + d(i,j)$, $j$ receives an update message from $i$ and sets $\lambda_j(D_i)$, $\lambda_j(W_i)$ and $\lambda_j(T_i)$ accordingly. In addition, $j$ resets $\Delta_j(i)$ to 0. After this time, $\Delta_j(i)$ is incremented every clock tick as long as $\lambda_j(D_i) + \Delta_j(i) < \alpha(\lambda_j(W_i) + d(i,j))$. Therefore, by time $t_e + t_1 + 2d(i,j) < t'$, it holds that $\lambda_j(D_i) + \Delta_j(i) \geq \lambda_j(W_i) + d(i,j)$. The rest follows from case I.

Case III: $i$ exhibited a join event at $t_0$ or afterwards, and $D_i(t) \geq W_i(t)$. Denote by $t_e \geq t_0$ the time of the last event at which $i$ became active or changed its weight. Following similar arguments as in case II, at $t_e + 2d(i,j) < t'$ it holds that $\Delta_j(i) \geq d(i,j)$, and hence $\lambda_j(D_i) + \Delta_j(i) \geq \lambda_j(W_i) + d(i,j)$. The rest follows from case I. ■