Self-Stabilizing Token Distribution with Constant-Space for Trees

Yuichi Sudo
Graduate School of Information Science and Technology, Osaka University, Japan
y-sudou@ist.osaka-u.ac.jp

Ajoy K. Datta
Department of Computer Science, University of Nevada, Las Vegas, USA
ajoy.datta@unlv.edu

Lawrence L. Larmore
Department of Computer Science, University of Nevada, Las Vegas, USA
lawrence.larmore@unlv.edu

Toshimitsu Masuzawa
Graduate School of Information Science and Technology, Osaka University, Japan
masuzawa@ist.osaka-u.ac.jp

Abstract

Self-stabilizing and silent distributed algorithms for token distribution in rooted tree networks are given. Initially, each process of a graph holds at most ℓ tokens. Our goal is to distribute the tokens in the whole network so that every process holds exactly k tokens. In the initial configuration, the total number of tokens in the network may not be equal to nk where n is the number of processes in the network. The root process is given the ability to create a new token or remove a token from the network. We aim to minimize the convergence time, the number of token moves, and the space complexity. A self-stabilizing token distribution algorithm that converges within $O(n\ell)$ asynchronous rounds and needs $\Theta(nh\epsilon)$ redundant (or unnecessary) token moves is given, where $\epsilon = \min(k, \ell - k)$ and h is the height of the tree network. Two novel ideas to reduce the number of redundant token moves are presented. One reduces the number of redundant token moves to $O(nh)$ without any additional costs while the other reduces the number of redundant token moves to $O(n)$, but increases the convergence time to $O(nh\ell)$. All algorithms given have constant memory at each process and each link register.

2012 ACM Subject Classification Theory of computation → Self-organization

Keywords and phrases token distribution, self-stabilization, constant-space algorithm

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.31

Related Version The brief announcement version of this paper is published in [16].

Acknowledgements This work was supported by JSPS KAKENHI Grant Numbers 17K19977 and 18K18000, and Japan Science and Technology Agency(JST) SICORP.

1 Introduction

The token distribution problem was originally defined by Peleg and Upfal in their seminal paper [15]. Consider a network of n processes and n tokens. Initially, the tokens are arbitrarily distributed among processes but with up to a maximum of ℓ tokens in any process. The problem is to distribute the tokens among the processes such that every process ends up with exactly one token. The above problem was redefined in another paper by the same


© Yuichi Sudo, Ajoy K. Datta, Lawrence L. Larmore, and Toshimitsu Masuzawa;
licensed under Creative Commons License CC-BY

authors [14] by considering m tokens instead of n tokens. The goal in this case is to reach a configuration at which there are either $\lfloor m/n \rfloor$ or $\lceil m/n \rceil$ tokens at each process. The token distribution problem is a form of the load balancing problem in distributed systems. A token can be considered to represent a unit of task (or load) of a process. The solution to the token distribution problem provides a solution of the load balancing problem, where the goal is to maintain the loads of process as evenly as possible.

The fault-tolerant (self-stabilizing) version of the problem of load balancing in distributed systems was first considered by Arora and Gouda [1]. A self-stabilizing algorithm has two properties, convergence and stability. The convergence property states that regardless of the initial number of tokens in the processes, when the faults cease to occur, that is, the environment no longer changes the load, the execution will reach in finite steps a state where the loads of processes are balanced. The stability property avoids the possibility of any execution shifting any unit of load between any two processes forever.

In [1] and many other papers on token distribution papers, a special property (referred to as *constraint* in [1]) is maintained. This property specifies that during any execution of the load balancing or token distribution algorithm, no new tokens are produced and no tokens are consumed by any process. However, that constraint cannot be maintained in our work due to the nature of the problem. The token distribution problem solved in this paper requires storing a fixed number (k) of tokens at each process. As we deal with self-stabilizing systems, the network (tree in this paper) can start in an arbitrary configuration where the total number of tokens in the network may not be exactly equal to nk. Instead, each process holds an arbitrary number, from zero to ℓ, of tokens in an initial configuration. Thus, our algorithm must make an exception to the constraint above. We assume that only the root process can push/pull tokens to/from the external store as needed.

We present three silent and self-stabilizing token distribution algorithms for rooted tree networks in this paper. The performances of the algorithms are summarized in Table 1. First, we present a self-stabilizing token distribution algorithm *Base*. This algorithm has the optimal convergence time, $O(n\ell)$ (asynchronous) rounds. (As we will see in Section 3, any self-stabilizing token distribution algorithm requires convergence time of $\Omega(n\ell)$ rounds.) However, *Base* may have a large number of redundant token moves; $\Theta(nh\epsilon)$ *redundant* (or unnecessary) token moves occur in the worst case where $\epsilon = \min(k, \ell - k)$, where h is the height of the tree network. Next, we combine the algorithm *Base* with a synchronizer or with PIF waves to reduce redundant token moves, which results in *SyncTokenDist* or *PIFTokenDist*, respectively. Algorithm *SyncTokenDist* reduces the number of redundant token moves to $O(nh)$ without additional costs, while *PIFTokenDist* drastically reduces the number of redundant token moves to the asymptotically optimal value, i.e., $O(n)$, at the expense of increasing convergence time from $O(n\ell)$ to $O(nh\ell)$ rounds. The work space space, i.e., the amount of memory needed to store information except for tokens, of all our algorithms are constant both per process and per link register.

1.1 Related Work

The token distribution problem is introduced by Peleg and Upfal in [15]. Another solution to the same problem is given by the same authors in [14]. In these papers, the problem is defined for general bounded degree graphs. Herley [8] gives another scheme to solve the problem and claims that his solution is more efficient if the time for the local computation steps is not ignored. Another version of the problem, called near-perfect token distribution problem is introduced in [2]. In this problem, at the termination of the algorithm, no more than $O(1)$ tokens can be present at any process. There is a variation of the near-perfect

Table 1 Token distribution algorithms for rooted trees. ($\epsilon = \min(k, \ell - k)$).

	Conv. Time	#Red. Token Moves	Work Space (Process,Link)
Base	$O(n\ell)$ rounds	$\Theta(nh\epsilon)$	$(0, O(1))$
SyncTokenDist	$O(n\ell)$ rounds	$O(nh)$	$(O(1), O(1))$
PIFTokenDist	$O(nh\ell)$ rounds	$O(n)$	$(O(1), O(1))$
LowerBounds	$\Omega(n\ell)$ rounds	$\Omega(n)$	-

token distribution problem, where the maximum difference of tokens between any pair of processes at the termination of the algorithm is non-constant. That is, the difference depends on some network parameter. Algorithms in [7, 10, 11] are in this category. Algorithms for general networks are given in [7], for complete binary trees in [10], and for meshes and torus networks in [11].

The token distribution problem for tree networks is stated, without any solution, by Peleg in [13]. In this paper, we solve the problem given in that paper. Another version of the token distribution problem on trees is stated in [12], where the total number of tokens, denoted by T, and the number of processes of the tree are computed by the algorithm. The tokens are then distributed perfectly among the processes in $O(Td)$ time, where d is the diameter of the tree. Although the asynchronous message passing model was used in their work, they used stronger communication primitives than the typical send/receive primitives. The solution in [9] is for tree networks, where processes use the knowledge of n. The *dimension-exchange* model used in [9] is also different from models used by the other algorithms discussed above.

The token distribution problem is a version of the load balancing problem. Arora and Gouda [1] give self-stabilizing load balancing algorithms for ring and tree networks. Starting from an arbitrary initial assignment of load (or tasks), their algorithms are guaranteed to converge to a state where the loads of any pair of processes differ by at most 1.

2 Preliminaries

2.1 Model of Computation

We consider a tree network $T = (V, E)$ where V is the set of n processes and E is the set of $n - 1$ links. Let v_{root} be the root process of T, and let $p(v)$ be the parent of any process $v \neq v_{\text{root}}$. Let $N(v)$ be the neighbors of a process v. Let $C(v)$ be the set of children of v, i.e., all neighbors of v except its parent. Let T_v be sub-tree of T consisting of v and its descendants. Let $n_v = |T_v|$. The height h_v of a process v is the length of the longest path through T between v and a leaf of T_v. Let $h = h_{v_{\text{root}}}$.

Each link $\{u, v\} \in E$ has two *link registers* (or just *registers*) $r_{u,v}$ and $r_{v,u}$. We call $r_{u,v}$ an *output register* of u and an *input register* of v. A process can read its input and its output registers. but can write only to its output registers. Thus, neighboring processes u and v can communicate with each other through $r_{u,v}$ and $r_{v,u}$.

A process v holds at most ℓ tokens, each of which is a bit sequence of length b. These tokens are stored in a dedicated memory space of the process, called the *internal token store*, written $v.\texttt{tokenStore}$, We use a link register to send and receive a token between processes. Each register $r_{u,v}$ contains at most one token in a dedicated variable $r_{u,v}.\texttt{token}$. The root process v_{root} can also access the *external token store*, which contains infinitely many tokens. As we will see later, v_{root} can reduce the total number of tokens in the tree, i.e., the tokens in the internal token stores and the links, by pushing a token into the external store, and can increase it by pulling a token from the external store.

We use the composite atomicity model with link registers. An algorithm $\mathcal{A}$ is specified by a set of local variables in processes, a set of shared variables in link registers, and actions that specify how a process v updates its local variables and shared variables in its output registers at each step, but not the local variables of its neighbors. When a process v executes the procedure, according to the values of its local variables, shared variables in $r_{u,v}$ and $r_{v,u}$ for all $u \in N(v)$, the number of tokens in its token store (i.e., $|v.\texttt{tokenStore}|$), and two given parameters ℓ and k, process v updates its local variables and shared variables of $r_{v,u}$, and executes `Push` and `Pull` arbitrarily many times. If a process v executes $v.\texttt{Push}(x)$, it append a token with bit sequence x into its token store; when it executes $v.\texttt{Pull}()$, it extracts and removes an arbitrary token from its token store. A process v can detect whether its local token store is empty (i.e., $|v.\texttt{tokenStore}| = 0$), and whether its local token store is full (i.e., $|v.\texttt{tokenStore}| = \ell$). A process v is not allowed to invoke `Pull` when its token store is empty, nor to invoke `Push` when its token store is full. The root process v_{root} can push a token into or pull a token out of the external store by executing $\texttt{exPush}(x)$ and $\texttt{exPull}()$, respectively. Typically, it can move a token from its local token store to the external store by $\texttt{exPush}(\texttt{Pull}())$ and can move a token in the reverse direction by $\texttt{Push}(\texttt{exPull}())$. There is one restriction, however, each of $\texttt{exPush}()$ and $\texttt{exPull}()$ can be executed at most once per step.[1] We say a process is *enabled* if execution of the procedure would change the value of at least one variable; otherwise, the process is *disabled*.

The values of all variables in a process define the *state* of the process; similarly, the values of the variables, including `token`, in a link register define the state of that register. A global state or *configuration* of T is specified by the the states of all processes, the states of all link registers, and the contents of the token stores of all processes. Let γ and γ' be two configurations of an algorithm $\mathcal{A}$ on T. We say that $\gamma \mapsto \gamma'$ is a *step* of $\mathcal{A}$ if there is a non-empty set $S \subseteq V$ of enabled processes such that γ changes to γ' when all the processes in S simultaneously execute the procedure of $\mathcal{A}$. A process in S is said to be *selected* at the step $\gamma \mapsto \gamma'$. We define an *execution* of $\mathcal{A}$ to be a maximal sequence $\gamma_0, \gamma_1, \ldots$ of configurations such that each $\gamma_i \mapsto \gamma_{i+1}$ is a step of $\mathcal{A}$. For any local variable `var1` and any shared variable `var2`, we denote local variable `var1` of a process v by $v.\texttt{var1}$, and shared variable `var2` of register $r_{u,v}$ by $r_{u,v}.\texttt{var2}$. Furthermore, for any configuration γ, we denote the values of $v.\texttt{var1}$ and $r_{u,v}.\texttt{var2}$ in configuration γ by $\gamma(v).\texttt{var1}$ and $\gamma(r_{u,v}).\texttt{var2}$, respectively.

We assume that a *scheduler* (or *daemon*) selects a set of processes that execute at each step of an execution. The execution ends when there are no enabled processes. In this paper, we assume the *distributed unfair daemon*, which selects an arbitrary nonempty subset of enabled processes at each step; it is *unfair* because it is not required to select a specific enabled process v unless v is the only enabled process.

2.2 Problem Specification

We now formally specify the problem. Given positive integers $k \leq \ell$, our goal is to reach a configuration where every process holds exactly k tokens starting from any configuration where the number of tokens at each process is arbitrary in the range $0, 1, \ldots, \ell$.

First, we give the definition of *legality* of an algorithm. Intuitively, we want to guarantee that no tokens in the network disappear from the network unless v_{root} pushes them into the external store, and that no new token appears in the network until v_{root} pulls a token from

[1] This is a natural restriction since we have the rule that a process can send at most one token to a given neighbor in a step, as we shall see later. Simply think of the external store as located in a fictitious neighboring process.

the external store. The nature of a token is defined by the application; we can assume it is a bit string. To transfer a token x from process u to v, process u writes x to $r_{u,v}$.`token`. But v cannot delete x from $r_{u,v}$.`token` since $r_{u,v}$ is an input register of v. Instead, we assume that every algorithm $\mathcal{A}$ specifies a predicate $\mathcal{P}_\mathcal{A}$ which indicates whether or not $\mathcal{A}$ recognizes that a token exists in a link register $r_{u,v}$ according to the states of $r_{u,v}$ and $r_{v,u}$. Let $R_\mathcal{A}$ be the set of all the states of a register in algorithm $\mathcal{A}$. Given $\mathcal{P}_\mathcal{A} : R_\mathcal{A} \times R_\mathcal{A} \to \{false, true\}$, we consider that a token represented by the bit sequence x exists in register $r_{u,v}$ if and only if $r_{u,v}$.`token` $= x$ and $\mathcal{P}_\mathcal{A}(a, b) = true$ where a and b are the states of $r_{u,v}$ and $r_{v,u}$, respectively. Then, the multiset of tokens in the network with a configuration γ is uniquely determined and we denote this multiset by $m_{\gamma,\mathcal{P}_\mathcal{A}}$. (i.e., A bit sequence x appears n_x times in multiset $m_{\gamma,\mathcal{P}_\mathcal{A}}$ if and only if exactly n_x tokens with contents x exists in the network.) We say that a step $\gamma \mapsto \gamma'$ is *legal* with $\mathcal{P}_\mathcal{A}$ if the following three conditions hold: (i) process v_{root} does not invoke both the two commands `exPush`(x) and `exPull`() in step $\gamma \mapsto \gamma'$, (ii) if v_{root} pulls a token x from (resp. pushes a token x to) the external store in step $\gamma \mapsto \gamma'$, the number of tokens with contents x increases (resp. decreases) by one and the numbers of other tokens are unchanged from $m_{\gamma,\mathcal{P}_\mathcal{A}}$ to $m_{\gamma',\mathcal{P}_\mathcal{A}}$, and (iii) if v_{root} does not execute either `exPush`(x) nor `exPull`() during step $\gamma \mapsto \gamma'$, then $m_{\gamma,\mathcal{P}_\mathcal{A}} = m_{\gamma',\mathcal{P}_\mathcal{A}}$. We say that an algorithm $\mathcal{A}$ is legal with $\mathcal{P}_\mathcal{A}$ if every step of algorithm $\mathcal{A}$ is legal with $\mathcal{P}_\mathcal{A}$.

We say that $\mathcal{A}$ is a self-stabilizing [6] token distribution algorithm if $\mathcal{A}$ is legal with some predicate $\mathcal{P}_\mathcal{A}$ and there is a set $\mathcal{L}_\mathcal{A}$ of *legitimate* configurations, such that the following three conditions are satisfied: (i) **Closure:** If $\gamma \in \mathcal{L}_\mathcal{A}$ holds, and $\gamma \mapsto \gamma'$ is a step of $\mathcal{A}$, then $\gamma' \in \mathcal{L}_\mathcal{A}$. (ii) **Convergence:** Every execution of $\mathcal{A}$, which starts from an arbitrary configuration, contains a configuration in $\mathcal{L}_\mathcal{A}$. (iii) **Correctness:** At any $\gamma \in \mathcal{L}_\mathcal{A}$, every process holds exactly k tokens in its token store and no registers hold tokens in γ in the sense of predicate $\mathcal{P}_\mathcal{A}$. We also say that a self-stabilizing token distribution algorithm $\mathcal{A}$ is *silent* if every execution of $\mathcal{A}$ is finite, that is, reaches a configuration where no process is enabled.

2.3 Complexities

We evaluate token distribution algorithms with three metrics – time complexity, space complexity, and the number of token moves.

We measure the time complexity in terms of (asynchronous) *rounds*. Let $\Gamma = \gamma_0, \gamma_1, \ldots$ be an execution of $\mathcal{A}$, and let $\mathcal{E}$ be the set of processes which are enabled at γ_0. We define the *first round* of execution Γ to be the smallest prefix, say $\gamma_0, \ldots, \gamma_t$, of Γ such that every member of $\mathcal{E}$ either executes or becomes disabled during the first t steps. We define $\gamma_0, \ldots \gamma_t$ to be the *range* of the first round of Γ. Let Γ' be the suffix of Γ starting from γ_t. The second round of Γ is defined to be the first round of Γ', and so forth.

It requires lb bits (resp. b bits) of space to manage a token store in each process (resp. variable `token` in each register). In this paper, we focus on *work space complexity* in each process and in each register of an algorithm. The work space complexity in each process (resp. in each register) is the bit length to represent all variables on the process (resp. in the register) except for `tokenStore` (resp. `token`).

Generally, a token is transferred from a process u to a process v in the following two steps: (i) u moves the token from u.`tokenStore` to $r_{u,v}$.`token`; (ii) v moves the token from $r_{u,v}$.`token` to v.`tokenStore`. In this paper, we regard the above two steps together as one token move and consider the number of token moves as the number of the occurrences of the former steps. Specifically, we say that a token moves from u to v in step $\gamma \mapsto \gamma'$ of algorithm $\mathcal{A}$ when predicate $\neg\mathcal{P}_\mathcal{A}(\gamma(r_{u,v}), \gamma(r_{v,u})) \wedge \mathcal{P}_\mathcal{A}(\gamma'(r_{u,v}), \gamma'(r_{v,u}))$ holds. Furthermore, introducing a *virtual process* $p(v_{\text{root}}) \notin V$, we also say that a token moves from v_{root} to

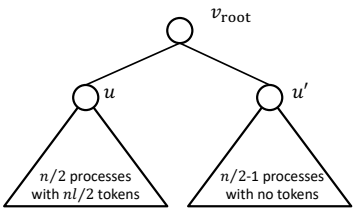

Figure 1 A tree to prove Theorem 1.

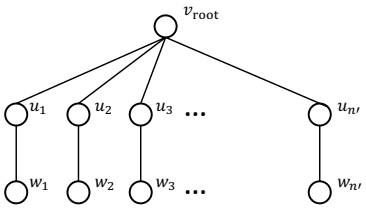

Figure 2 A tree to prove Theorem 2.

$p(v_{\text{root}})$ (resp. from $p(v_{\text{root}})$ to v_{root}) in a step when v_{root} invokes `exPush()` (resp. `exPull()`) in that step. We are interested in the number of redundant token moves. Given $v \in V$ and configuration γ, let $\tau(\gamma, v)$ be the number of tokens in input registers of v in configuration γ. Then, we define $\Delta(\gamma, v) = \sum_{u \in T_v} d(\gamma, u)$ where $d(\gamma, u) = |\gamma(u).\texttt{tokenStore}| + \tau(\gamma, u) - k$. Intuitively, $\Delta(\gamma, v)$ is the number of tokens that v must send to $p(v)$ to achieve the token distribution in an execution starting from configuration γ if $\Delta(\gamma, v) \geq 0$; Otherwise, $p(v)$ must send $-\Delta(\gamma, v)$ tokens to v in the execution. Therefore, in an execution $\Gamma = \gamma_0, \gamma_1, \ldots$, we need at least $\sum_{v \in V} |\Delta(\gamma_0, v)|$ token moves to achieve token distribution. We define *the number of redundant token moves* in $\Gamma = \gamma_0, \gamma_1, \ldots$ as the total number of token moves in the execution minus $\sum_{v \in V} |\Delta(\gamma_0, v)|$.

3 Lower Bounds

▶ **Theorem 1.** *For any self-stabilizing token distribution algorithm $\mathcal{A}$, there exists an execution of $\mathcal{A}$ that takes $\Omega(nl)$ rounds to reach a configuration where all processes hold exactly k tokens.*

Proof. Consider a tree network of an even number of processes where v_{root} has two children u and u' where $n_u = n/2$ and $n_{u'} = n/2 - 1$, and consider a configuration of $\mathcal{A}$ on the graph where T_u holds $n_u \cdot l$ tokens in total and $T_{u'}$ holds zero tokens (Figure 1). Obviously, process u must be selected by the scheduler $\Omega(n(l-k))$ times to send $n_u(l-k)$ tokens to v_{root} and process u' must be selected by the scheduler $\Omega(nk)$ times to receive $n_v \cdot k$ tokens from v_{root}. Thus, there exists an execution of $\mathcal{A}$ starting from this configuration which takes $\max(\Omega(n(l-k)), \Omega(nk)) = \Omega(nl)$ rounds to achieve the token distribution. ◀

▶ **Theorem 2.** *For any self-stabilizing token distribution algorithm $\mathcal{A}$, there exists an execution of $\mathcal{A}$ such that the number of redundant token moves in the execution is $\Omega(n)$.*

Proof. Consider the tree network where v_{root} has n' children $u_1, u_2, \ldots, u_{n'}$, each u_i has exactly one child w_i, and no other processes exist (See Figure 2). Consider an execution of $\mathcal{A}$, on this graph, that starts from a configuration where, for all $i = 1, 2, \ldots, n'$, process u_i holds $k+1$ tokens, w_i holds no token, and u_i, $r_{u_i, v_{\text{root}}}$, r_{v_{root}, u_i}, r_{u_i, w_i}, and r_{w_i, u_i} are in states such that u_i sends a token to v_{root} when u_i is selected by the scheduler. (Such states must exist for any self-stabilizing token distribution algorithm.) If every u_i is selected by the scheduler in the first step of this execution, then $n' = \lfloor n/2 \rfloor$ redundant token moves must happen in the first step. ◀

4 Constant-Space Algorithms for Self-stabilizing Token Distribution

Before presenting the three algorithms – *Base*, *SyncTokenDist*, and *PIFTokenDist* – we describe the token-passing mechanism (or handshake mechanism) that all these algorithms use to send and receive a token between two processes. Each register $r_{u,v}$ has two shared boolean variables $r_{u,v}.\texttt{exist}$ and $r_{u,v}.\texttt{ready}$ in addition to $r_{u,v}.\texttt{token}$. We use predicate $\mathcal{P}_{Base}(a,b) \equiv a.\texttt{exist} \wedge b.\texttt{ready}$ (where a and b are states of $r_{u,v}$ and $r_{v,u}$, respectively) for all the three algorithms to represent the existence of a token in link register: register $r_{u,v}$ holds a token when $r_{u,v}.\texttt{exist} \wedge r_{v,u}.\texttt{ready}$ holds. Let u and v be neighboring processes. We define four predicates used in the pseudocodes as follows:

$$\begin{aligned}
\mathit{IsReady}(u,v) &\equiv (\neg r_{u,v}.\texttt{exist} \wedge r_{v,u}.\texttt{ready}),\\
\mathit{JustSent}(u,v) &\equiv (r_{u,v}.\texttt{exist} \wedge r_{v,u}.\texttt{ready}),\\
\mathit{JustReceived}(u,v) &\equiv (r_{u,v}.\texttt{exist} \wedge \neg r_{v,u}.\texttt{ready}),\\
\mathit{NotReady}(u,v) &\equiv (\neg r_{u,v}.\texttt{exist} \wedge \neg r_{v,u}.\texttt{ready}).
\end{aligned}$$

All the three algorithms adopt the token passing mechanism consisting of the following rules to send a token from a process u to a process v:

Rule 1 When $\mathit{IsReady}(u,v)$ and $|u.\texttt{tokenStore}| > 0$, process u may perform "$r_{u,v}.\texttt{token} \leftarrow \texttt{Pull}()$" and "$r_{u,v}.\texttt{exist} \leftarrow \mathit{true}$". After that, $\mathit{JustSent}(u,v)$ becomes true.

Rule 2 When $\mathit{JustSent}(u,v)$ and $|v.\texttt{tokenStore}| < l$, process v may perform "$\texttt{Push}(r_{u,v}.\texttt{token})$" and "$r_{v,u}.\texttt{ready} \leftarrow \mathit{false}$". After that, $\mathit{JustReceived}(u,v)$ becomes true.

Rule 3 When $\mathit{JustReceived}(u,v)$, process u may perform "$r_{u,v}.\texttt{exist} \leftarrow \mathit{false}$". After that, $\mathit{NotReady}(u,v)$ becomes true.

Rule 4 When $\mathit{NotReady}(u,v)$, process v may perform "$r_{v,u}.\texttt{ready} \leftarrow \mathit{true}$". After that, $\mathit{IsReady}(u,v)$ becomes true.

By definition of $\mathcal{P}_{Base}$, a token exists in $r_{u,v}$ if and only if $\mathit{JustSent}(u,v)$ is true. In any step $\gamma \mapsto \gamma'$, none of the above rule changes $m_{\gamma,\mathcal{P}_{Base}}$, i.e., a multiset of tokens in the network. In addition to the above rules, there is another mechanism to update a local token store. The root process v_{root} pushes a token to and pulls a token from the external store. Specifically, v_{root} pulls a token from the external store by $\texttt{Push}(\texttt{exPull}())$, and v_{root} pushes a token to the external store by "$\texttt{exPush}(\texttt{Pull}())$". The former (resp. latter) operation is performed only when $|v_{\mathrm{root}}.\texttt{tokenStore}| < l$ (resp. $|v_{\mathrm{root}}.\texttt{tokenStore}| > 0$). Shared variables $\texttt{token}$, $\texttt{exist}$, and $\texttt{ready}$ are never updated in any way other than Rules 1-4.

An arbitrary step $\gamma \mapsto \gamma'$ of an algorithm which adopts the above token mechanism is legal because Rules 1-4 do not affect the multiset $m_{\gamma',\mathcal{P}_{Base}}$. Thus the following lemma holds.

▶ **Lemma 3.** *Algorithms Base, SyncTokenDist, and PIFTokenDist are all legal with* $\mathcal{P}_{Base}$.

4.1 Algorithm *Base*

In this section, we define our self-stabilizing token distribution algorithm, which we call *Base*. The work space complexity of *Base* in process (resp. on register) is zero (resp. constant). Its convergence time is $O(n\ell)$ rounds and it makes $\Theta(nh\epsilon)$ redundant token moves.

Some functions take a configuration (e.g., γ) as one of their arguments, such as $\Delta(\gamma, v)$, defined in Section 2.2. In what follows, we omit the configuration when it is clear from the context. For example, we write $\Delta(v)$ instead of $\Delta(\gamma, v)$. We use the standard notation $\mathrm{sgn}(x)$, that is, $\mathrm{sgn}(x) = 1$, $\mathrm{sgn}(x) = 0$, and $\mathrm{sgn}(x) = -1$ if $x > 0$, $x = 0$, and $x < 0$, respectively.

Algorithm 1 *Base*

[Variables of process v and register $r_{v,u}$]

v.`tokenStore` {an array containing at most l tokens}
$r_{v,u}$.`exist`, $r_{v,u}$.`ready`
$r_{v,u}$.`est` $\in \{1, 0^+, 0, 0^-, -1, \bot\}$ {$r_{v,u}$.`est` $\in \{1, 0, -1\}$ if v is a leaf.}
$r_{v,u}$.`token` $\in 2^b$

[Actions of process v]

1: $r_{v,u}$.`exist` $\leftarrow$ *false* for all $u \in N(v)$ such that *JustReceived*(v, u)
2: $r_{v,u}$.`ready` $\leftarrow$ *true* for all $u \in N(v)$ such that *NotReady*(u, v)
3: `ReceiveToken`(u) for all $u \in N(v)$ such that *JustSent*(u, v)
4: `SendToken`(u) for all $u \in C(v)$ such that $r_{u,v}$.`est` $= -1 \wedge$ *IsReady*(v, u)
5: **if** $v = v_{\text{root}}$ **then**
6: `AdjustTokens`()
7: **else**
8: `SendToken`$(p(v))$ **if** *Est*$(v) = 1 \wedge$ *IsReady*$(v, p(v))$
9: $r_{v,p(v)}$.`est` $\leftarrow$ *Est*(v)
10: **end if**

[`SendToken`(u)]:

11: **if** $|v$.`tokenStore`$| > 0$ **then**
12: $r_{v,u}$.`token` $\leftarrow$ `Pull`()
13: $r_{v,u}$.`exist` $\leftarrow$ *true*
14: `ReceiveToken`(w) **if** $\exists w \in N(v)$: *JustSent*(w, v)
15: **end if**

[`ReceiveToken`(u)]:

16: **if** $|v$.`tokenStore`$| < l$ **then**
17: `Push`$(r_{u,v}$.`token`$)$
18: $r_{v,u}$.`ready` $\leftarrow$ *false*
19: **end if**

[`AdjustTokens`()]:

20: **if** *Est*$(v) = -1$ **then**
21: `Push`(`exPull`())
22: **else if** *Est*$(v) = 1$ **then**
23: `exPush`(`Pull`())
24: `ReceiveToken`(u) **if** $\exists u \in N(v)$: *JustSent*(u, v)
25: **end if**

During an execution of *Base*, each process v tries to estimate $\text{sgn}(\Delta(v))$, that is, tries to find whether $\Delta(v)$ is positive, negative, or just zero. Each process $v \neq v_{\text{root}}$ reports that estimate to its parent $p(v)$ using a shared variable $r_{v,p(v)}$.`est`. When its estimate is negative, $p(v)$ sends a token to v if $p(v)$ holds a token and *IsReady*$(p(v), v)$. When the estimate is positive, v sends a token to its parent $p(v)$ if v holds a token and *IsReady*$(v, p(v))$. The root v_{root} always pulls a new token from the external store to increase $\Delta(v_{\text{root}})$ when its estimate is negative, and pushes a token to the external store to decrease $\Delta(v_{\text{root}})$ when the estimate is positive. If all processes v correctly estimate $\text{sgn}(\Delta(v))$, each of them eventually holds k tokens. After that, no process sends a token.

Thus, estimating $\mathrm{sgn}(\Delta(v))$ is the key of algorithm *Base*. Each process v computes $Est(v)$, its estimate of $\mathrm{sgn}(\Delta(v))$ as follows.

$$Est(v) = \begin{cases} 1 & (Diff(v) > 0 \wedge \forall u \in C(v) : r_{u,v}.\texttt{est} \in \{1, 0^+, 0\}) \\ 0^+ & (Diff(v) = 0 \wedge \forall u \in C(v) : r_{u,v}.\texttt{est} \in \{1, 0^+, 0\} \\ & \qquad \wedge \exists w \in C(v) : r_{u,v}.\texttt{est} \in \{0^+, 1\}) \\ 0 & (Diff(v) = 0 \wedge \forall u \in C(v) : r_{u,v}.\texttt{est} = 0) \\ 0^- & (Diff(v) = 0 \wedge \forall u \in C(v) : r_{u,v}.\texttt{est} \in \{-1, 0^-, 0\} \\ & \qquad \wedge \exists w \in C(v) : r_{u,v}.\texttt{est} \in \{0^-, -1\}) \\ -1 & (Diff(v) < 0 \wedge \forall u \in C(v) : r_{u,v}.\texttt{est} \in \{-1, 0^-, 0\}) \\ \bot & (\text{otherwise}), \end{cases}$$

$$Diff(v) = |v.\texttt{tokenStore}| + |\{u \in N(v) \mid JustSent(u, v)\}| - k$$

where the candidate values 1, 0^+, 0, 0^-, -1, and $\bot$ of $Est(v)$ represent that the estimate is positive, "never negative", zero, "never positive", negative, and "unsure", respectively. A process sends a token to its parent only when its estimate is 1, and it sends a token to its child only when the child's estimate is -1. Note that $\Delta(v) = \sum_{u \in T_v} Diff(u)$. For process $v \in V \setminus \{v_{\mathrm{root}}\}$, the domain of variable $r_{v,p(v)}.\texttt{est}$ is $\{1, 0^+, 0, 0^-, -1\}$ if v is not a leaf, and is $\{1, 0, -1\}$ if v is a leaf.

Algorithm 1 gives the pseudocode of *Base*. When v executes, it first updates $r_{v,u}.\texttt{exist}$ and $r_{v,u}.\texttt{ready}$ for all $u \in N(v)$ such that $JustReceived(v, u)$ or $NotReady(u, v)$, according to the token-passing mechanism (Lines 1-2). Then, v checks whether each input register $r_{u,v}$ holds a token, and receives that token by invoking $\texttt{ReceiveToken}(u)$ if its token store is not full, using the token-passing mechanism (Lines 3, 16-19). Then, v sends a token to each $u \in C(v)$ such that $IsReady(v, u)$ and $r_{u,v}.\texttt{est} = -1$ (i.e., $\Delta(u)$ is estimated to be negative) by invoking $\texttt{SendToken}(u)$ if v's token store is not empty (Lines 4, 11-15). Next, $v \in V \setminus \{v_{\mathrm{root}}\}$ and v_{root} perform different action. If $v \neq v_{\mathrm{root}}$, it sends a token to its parent if $IsReady(v, p(v))$ and $Est(v) = 1$, i.e., $\Delta(v)$ is estimated to be positive, and reports the latest estimate of $\mathrm{sgn}(\Delta(v))$ to its parent, and stores $Est(v)$ into $r_{v,p(v)}.\texttt{est}$ (Lines 7-10). Note that $Diff(v)$ decreases when v sends a token to its parent, hence the value of $Est(v)$ may differ in Lines 8 and 9. Process v_{root} invokes $\texttt{AdjustTokens}()$ by which v_{root} may increase or decrease the number of tokens in the tree by pulling a token from or pushing a token to the external store (Lines 6, 20-25).

In the algorithm description, we have used several functions which take a process as an argument, like $Est(v)$ and $Diff(v)$. The values of such functions depend on the states of the process and its registers, i.e., they depend on the current configuration. In what follows, we sometimes denote such functions with an explicitly specified configuration. For example, we sometimes denote $Est(v)$ in configuration γ by $Est(\gamma, v)$ and denote $Diff(v)$ in configuration γ by $Diff(\gamma, v)$.

In what follows, we show the correctness, the number of redundant token moves, and the convergence time of algorithm *Base*. First, we show that every execution of *Base* is finite (Lemma 4). The correctness of *Base* immediately follows. (Theorem 5).

▶ **Lemma 4.** *Every execution of Base is finite.*

Proof. Fix an execution $\Gamma = \gamma_0, \gamma_1, \ldots$ of *Base*. For any $v \in V$, we define a predicate $P_{\mathrm{finite}}(v) \equiv P_{\mathrm{finiteMove}}(v) \wedge P_{\mathrm{finiteEst}}(v)$, where $P_{\mathrm{finiteMove}}(v)$ and $P_{\mathrm{finiteEst}}(v)$ are also predicates: $P_{\mathrm{finiteMove}}(v)$ holds if and only if v sends and receives tokens only finitely many times in Γ, and $P_{\mathrm{finiteEst}}(v)$ holds if and only if $v = v_{\mathrm{root}}$ or v changes the value

of $r_{v,p(v)}.\texttt{est}$ only finitely many times in Γ. In the remainder of this proof, we prove $(\forall u \in C(v); P_{\text{finite}}(u)) \Rightarrow P_{\text{finite}}(v)$ for all $v \in V$. This proposition guarantees that (i) every leaf w satisfies $P_{\text{finite}}(w)$ because it has no children, and (ii) every process v with height $i > 0$ satisfies $P_{\text{finite}}(v)$ if every process u with height $i-1$ satisfies $P_{\text{finite}}(u)$. Thus, $P_{\text{finite}}(v')$ holds for all v by induction on the height h_v of v, from which gives the lemma.

Let $v \in V$, and suppose $P_{\text{finite}}(u)$ holds for all $u \in C(v)$. Then, v sends or receives no token to or from its children, and $r_{u,v}.\texttt{est}$ remains unchanged for all $u \in C(v)$ after some point of the execution. Let $\gamma_t \mapsto \gamma_{t+1}$ be the first step that v is selected after that point. (The unfair daemon may never selects v after that point, but we need not consider this case because then $P_{\text{finite}}(v)$ clearly holds.) In an execution after that step, say $\Gamma_{t+1} = \gamma_{t+1}, \gamma_{t+2}, \ldots$, process v sends a token to $p(v)$ only if $\mathit{Diff}(v) > 0$, and $p(v)$ sends a token to v only if $\mathit{Diff}(v) < 0$. Hence, v sends and receives tokens at most $|\mathit{Diff}(\gamma_{t+1}, v)|$ times in Γ_{t+1}, which implies $P_{\text{finiteMove}}(v)$. Thus, there exists $t' > t$ such that v never sends or receives a token after $\gamma_{t'}$. Since $\mathit{Diff}(v)$ never changes after $\gamma_{t'}$, $\mathit{Est}(v)$ also never changes after $\gamma_{t'}$. This means that $r_{v,p(v)}.\texttt{est}$ never changes after $\gamma_{t'+1}$, which implies $P_{\text{finiteEst}}(v)$. Thus $P_{\text{finite}}(v)$ holds in all cases. ◀

▶ **Theorem 5.** *Algorithm Base is a silent self-stabilizing token distribution algorithm.*

Proof. Let $\Gamma = \gamma_0, \gamma_1, \ldots$ be an execution of *Base*. Lemma 4 guarantees that γ ends at its final configuration γ_f, that is, $\Gamma = \gamma_0, \gamma_1, \ldots, \gamma_f$. No process is enabled in γ_f. Hence, every process holds k tokens in its token store and no token exists in registers in γ_f because otherwise some process must be enabled. ◀

We now give an asymptotically tight bound. on the number of redundant token moves of *Base*. Recall that $\epsilon = \min(k, l-k)$.

▶ **Lemma 6.** *The number of redundant token moves in any execution of Base is $O(nh\epsilon)$.*

Proof. For any process v, let $\Delta^+(v) = \sum_{u \in T_v} \max(\mathit{Diff}(u), 0)$ and $\Delta^-(v) = \sum_{u \in T_v} \max(-\mathit{Diff}(u), 0)$. During the execution $\Gamma = \gamma_0, \gamma_1, \ldots$ of *Base*, each v sends a token to $p(v)$ at most $\Delta^+(\gamma_0, v) + n_v$ times. This is because $\Delta^+(v)$ is monotonically non-increasing, except for the case that the parent of a process $u \in T_v$ sends a token to u before the first time u is selected by the scheduler, and $\Delta^+(v)$ decrements by one every time v sends a token to $p(v)$. Similarly, $p(v)$ sends a token to v at most $\Delta^-(\gamma_0, v)$ times in Γ. Since $\Delta(\gamma_0, v) = \Delta^+(\gamma_0, v) - \Delta^-(\gamma_0, v)$ and $\sum_{v \in V} n_v \leq nh$, the number of redundant token moves in Γ is at most $\sum_{v \in V}(\Delta^+(\gamma_0, v) + \Delta^-(\gamma_0, v) + n_v) - \sum_{v \in V} |\Delta(\gamma_0, v)| = \sum_{v \in V}(2\min(\Delta^+(\gamma_0, v), \Delta^-(\gamma_0, v)) + n_v) \leq \sum_{v \in V}(2\min(n_v(l-k), n_v \cdot k) + n_v) = O(nh\epsilon)$. ◀

▶ **Lemma 7.** *The number of redundant token moves in an execution of Base is $\Omega(nh\epsilon)$.*

Proof. Consider the network shown in Figure 3 where $2x \cdot 2y$ processes except for v_{root} are divided into four sets A, B, C, and D, each of which consists of xy processes. Consider a configuration where each process v in $\mathrm{A} \cup \mathrm{D}$ holds no token (i.e., $\mathit{Diff}(v) = -k$), each process u in $\mathrm{B} \cup \mathrm{C}$ holds ℓ tokens (i.e., $\mathit{Diff}(u) = \ell - k$), and $r_{w_i,p(w_i)}.\texttt{est} = r_{w'_i,p(w'_i)} = 0$ holds for all $i = 1, 2, \ldots, y$. In an execution starting from the configuration, the scheduler can select processes only in $\mathrm{A} \cup \mathrm{D} \cup \{v_{\text{root}}\}$ until all processes in $\mathrm{A} \cup \mathrm{D} \cup \{v_{\text{root}}\}$ become disabled. Such an execution must make at least $\min(x(x-1)yk/2, x(x-1)y(l-k)/2) = \Omega(nh\epsilon)$ redundant token moves. ◀

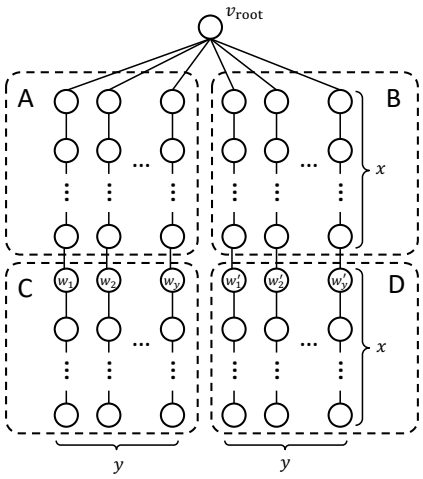

Figure 3 A tree to prove Lemma 7.

Finally, we analyze the convergence time of *Base*. We first consider the values of $r_{v,p(v)}.\texttt{est}$. Specifically, we prove that every execution of *Base* satisfies predicate $P_{\text{est}}(v)$, which is defined as follows, within $2h_v$ rounds, for any process v other than v_{root}:

$$
\begin{aligned}
P_{\text{est}}(v) &\equiv P_1(v) \wedge P_2(v) \wedge P_3(v) \wedge P_4(v) \wedge Q_1(v) \wedge Q_2(v) \wedge Q_3(v) \wedge \forall u \in C(v) : P_{\text{est}}(u) \\
P_1(v) &\equiv r_{v,p(v)}.\texttt{est} \in \{1, 0^+\} \;\Rightarrow\; \Delta(v) > 0 \wedge \mathit{Diff}(v) \geq 0 \\
P_2(v) &\equiv r_{v,p(v)}.\texttt{est} = 0 \;\Rightarrow\; \Delta(v) = 0 \wedge \mathit{Diff}(v) = 0 \\
P_3(v) &\equiv r_{v,p(v)}.\texttt{est} = 0^- \;\Rightarrow\; \Delta(v) \leq 0 \wedge \mathit{Diff}(v) \leq 0 \\
P_4(v) &\equiv r_{v,p(v)}.\texttt{est} = -1 \;\Rightarrow\; (\Delta(v) < 0 \vee (\Delta(v) = 0 \wedge \mathit{JustSent}(p(v), v))) \\
&\qquad\qquad \wedge \mathit{Diff}(v) \leq 0, \\
Q_1(v) &\equiv r_{v,p(v)}.\texttt{est} \in \{1, 0^+\} \;\Rightarrow\; \forall u \in T_v : r_{u,p(u)}.\texttt{est} \in \{1, 0^+, 0\}, \\
Q_2(v) &\equiv r_{v,p(v)}.\texttt{est} = 0 \;\Rightarrow\; \forall u \in T_v : r_{u,p(u)}.\texttt{est} = 0, \\
Q_3(v) &\equiv r_{v,p(v)}.\texttt{est} \in \{0^-, -1\} \;\Rightarrow\; \forall u \in T_v : r_{u,p(u)}.\texttt{est} \in \{0, 0^-, -1\}.
\end{aligned}
$$

▶ **Lemma 8.** *Let $v \in V \setminus \{v_{\text{root}}\}$ and let $\gamma \mapsto \gamma'$ be a step of Base where $P_{\text{est}}(\gamma, v)$ holds. Then, the following three statements hold:*

$$\gamma(r_{v,p(v)}).\texttt{est} = 0 \Rightarrow \gamma'(r_{v,p(v)}).\texttt{est} = 0 \tag{1}$$

$$\gamma(r_{v,p(v)}).\texttt{est} \in \{1, 0^+\} \Rightarrow \gamma'(r_{v,p(v)}).\texttt{est} \in \{1, 0^+, 0\} \tag{2}$$

$$\gamma(r_{v,p(v)}).\texttt{est} \in \{0^-, -1\} \Rightarrow \gamma'(r_{v,p(v)}).\texttt{est} \in \{0, 0^-, -1\} \tag{3}$$

Proof. If $\gamma(r_{v,p(v)}).\texttt{est} = 0$, then $\gamma'(r_{v,p(v)}).\texttt{est} = 0$ because $P_{\text{est}}(\gamma, v)$ implies that $\mathit{Diff}(\gamma, v) = 0$ and $\gamma(r_{u,v}).\texttt{est} = 0$ for all $u \in C(v)$. If $\gamma(r_{v,p(v)}).\texttt{est} \in \{1, 0^+\}$, then $\gamma'(r_{v,p(v)}).\texttt{est} \in \{1, 0^+, 0\}$ because $P_{\text{est}}(\gamma, v)$ implies that $\mathit{Diff}(\gamma, v) \geq 0$ and $\forall u \in C(v) : \gamma(r_{u,v}).\texttt{est} \in \{1, 0^+, 0\}$. Similarly, $\gamma'(r_{v,p(v)}).\texttt{est} \in \{0, 0^-, -1\}$ holds if $\gamma(r_{v,p(v)}).\texttt{est} \in \{0^-, -1\}$. ◀

▶ **Lemma 9.** *Let $v \in V$. If $P_{\text{est}}(u)$ holds for all $u \in C(v)$, then once $P_{\text{est}}(v)$ holds, $P_{\text{est}}(v)$ always holds.*

Proof. Let $\gamma \mapsto \gamma'$ be a step of *Base* where $P_{\mathrm{est}}(\gamma, v)$ holds and $P_{\mathrm{est}}(\gamma, u)$ and $P_{\mathrm{est}}(\gamma', u)$ hold for all $u \in C(v)$. It suffices to show $P_{\mathrm{est}}(\gamma', v)$ to prove the lemma. First, consider the case of $\gamma(r_{v,p(v)}).\mathtt{est} = \bot$. If $r_{v,p(v)}.\mathtt{est}$ remains $\bot$ in step $\gamma \mapsto \gamma'$, then $P_{\mathrm{est}}(\gamma', v)$ trivially holds. If $r_{v,p(v)}.\mathtt{est}$ becomes 1 or 0^+ at the step, i.e., $\gamma'(r_{v,p(v)}.\mathtt{est}) \in \{1, 0^+\}$, then $P_1(\gamma', v)$ holds and $\gamma(r_{u,v}.\mathtt{est}) \in \{1, 0^+, 0\}$ for all $u \in C(v)$ by the definition of *Base*. The latter statement, and Lemma 8, imply $Q_1(\gamma', v)$. Hence, $P_{\mathrm{est}}(\gamma', v)$ holds if $\gamma'(r_{v,p(v)}.\mathtt{est}) \in \{1, 0^+\}$. Similarly, $P_{\mathrm{est}}(\gamma', v)$ holds if $\gamma'(r_{v,p(v)}.\mathtt{est}) \in \{0, 0^-, -1\}$. Next, suppose $\gamma(r_{v,p(v)}).\mathtt{est} \neq \bot$. In this case, $\gamma'(r_{v,p(v)}).\mathtt{est} \neq \bot$ holds by Lemma 8. If $\gamma'(r_{v,p(v)}).\mathtt{est} \in \{1, 0^+\}$, then $P_1(\gamma', v)$ trivially holds by the definition of *Base*, and $P_{\mathrm{est}}(\gamma, v)$ and Lemma 8 imply that $\gamma'(r_{u,v}).\mathtt{est} \in \{1, 0^+, 0\}$ for all $u \in C(v)$, which implies $Q_1(\gamma', v)$. Hence, $P_{\mathrm{est}}(\gamma', v)$ holds if $\gamma'(r_{v,p(v)}.\mathtt{est}) \in \{1, 0^+\}$. Similarly, $P_{\mathrm{est}}(\gamma', v)$ holds if $\gamma'(r_{v,p(v)}.\mathtt{est}) \in \{0, 0^-, -1\}$. ◀

▶ **Lemma 10.** *Let $v \in V$. If $P_{\mathrm{est}}(u)$ holds for all $u \in C(v)$, $P_{\mathrm{est}}(v)$ holds once v is selected twice by the scheduler or v is disabled.*

Proof. Consider an execution $\Gamma = \gamma_0, \gamma_1, \ldots$ of *Base* where $P_{\mathrm{est}}(\gamma_t, u)$ holds for all $t \geq 0$ and all $u \in C(v)$. During the execution, $P_{\mathrm{est}}(v)$ holds when v is disabled, by the definition of algorithm *Base*, and by the assumption that $P_{\mathrm{est}}(u)$ holds for all $u \in C(v)$. Hence, it suffices to show that $P_{\mathrm{est}}(\gamma_{t+1}, v)$ or $P_{\mathrm{est}}(\gamma_{t'+1}, v)$ holds where $\gamma_t \mapsto \gamma_{t+1}$ and $\gamma_{t'} \mapsto \gamma_{t'+1}$ are the first and the second steps where v is selected by the scheduler in Γ. Due to the token passing mechanism, $p(v)$ cannot send a token to $r_{p(v),v}$ in both steps. For any step $\gamma \mapsto \gamma'$, $P_{\mathrm{est}}(\gamma', v)$ holds by the definition of *Base* if v is selected by the scheduler in $\gamma \mapsto \gamma'$, $p(v)$ cannot send a token to $r_{p(v),v}$ at the step, and $P_{\mathrm{est}}(\gamma, u) \wedge P_{\mathrm{est}}(\gamma', u)$ holds for all $u \in C(v)$. Hence, $P_{\mathrm{est}}(\gamma_{t+1}, v)$ or $P_{\mathrm{est}}(\gamma_{t'+1}, v)$ holds. ◀

▶ **Lemma 11.** *Let $v \in V$ and let $\Gamma = \gamma_0, \gamma_1, \ldots$ be an execution of Base. The predicate $P_{\mathrm{est}}(\gamma_t, v)$ holds for any $t \geq s(\Gamma, 2h_v)$, i.e., $P_{\mathrm{est}}(v)$ always holds after $2h_v$ rounds have elapsed.*

Proof. By induction on h_v. ◀

We define $\mathcal{C}_{\mathrm{est}}$ to be the set of all configurations where $P_{\mathrm{est}}(v)$ holds for all $v \neq v_{\mathrm{root}}$. The following corollary directly follows from Lemma 9 and Lemma 10. Furthermore, Lemma 13, below trivially holds by the definition of *Base*. Note that Lemma 13 implies that no redundant token move can occur after an execution reaches a configuration in $\mathcal{C}_{\mathrm{est}}$.

▶ **Corollary 12.** *An execution of Base reaches a configuration of $\mathcal{C}_{\mathrm{est}}$ within $2h$ rounds and never deviates from $\mathcal{C}_{\mathrm{est}}$ thereafter.*

▶ **Lemma 13.** *For any process v, $|\Delta(v)|$ is monotonically non-increasing during an execution of Base starting from a configuration in $\mathcal{C}_{\mathrm{est}}$.*

In the rest of this section, we prove that every execution starting from a configuration in $\mathcal{C}_{\mathrm{est}}$ finishes in $O(n\ell)$ rounds. Given $v \in V$, we define

$$R(v) = \min\left\{c \geq 0 \;\middle|\; \forall j = 1, 2, \ldots, |\Delta(v)| : \textstyle\sum_{u \in C_{2(j+c)-1}(v)} |\mathit{Diff}(u)| \geq j\right\}$$

where $C_i(v)$ is the set of processes in T_v whose distance from v is no greater than i (e.g., $C_0(v) = \{v\}$ and $C_1(v) = \{v\} \cup C(v)$). See Figure 4. In the left tree of the figure, $R(v) = 1$ because $C_{2j+1}(v)$ holds at least j extra tokens (i.e., $\sum_{u \in C_{2j+1}(v)} \mathit{Diff}(v) \geq j$) for all $j = 1, 2, 3$, but C_1 holds no extra token. In the right tree of the figure, $R(v) = 3$ because $C_{2j+5}(v)$ is short at least j tokens (i.e., $\sum_{u \in C_{2j+5}(v)}(-\mathit{Diff}(v)) \geq j$) for $j = 1, 2, 3$, but C_7 is short only one token. Let $\Gamma = \gamma_0, \gamma_1, \ldots$ be an execution starting from a

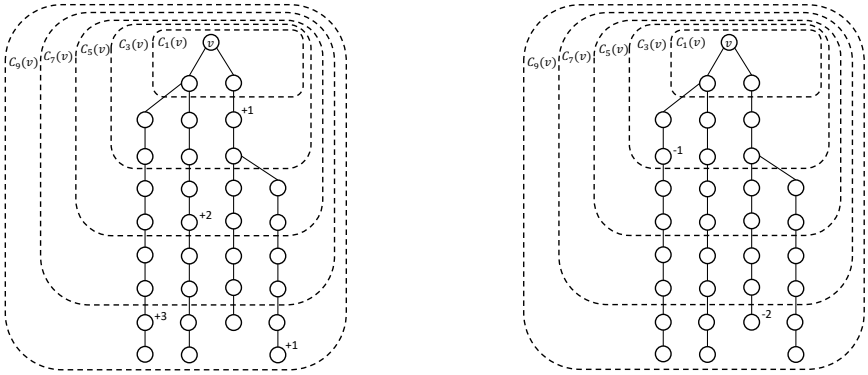

Figure 4 A tree network T_v with $\Delta(v) > 0$ (Left) A tree network with $\Delta(v) < 0$ (Right). The signed integer with process w indicates $\mathit{Diff}(w)$, omitted when $\mathit{Diff}(w) = 0$.

configuration in $\mathcal{C}_{\mathrm{est}}$ and $v \in V \setminus \{v_{\mathrm{root}}\}$ be a process such that $r_{v,p(v)}.\mathtt{est} \neq \perp$ holds for some configuration γ_t. Then either $\forall u \in T_v : r_{u,p(u)}.\mathtt{est} \in \{1, 0^+, 0\}$ or $\forall u \in T_v : r_{u,p(u)}.\mathtt{est} \in \{0, 0^-, -1\}$ holds for $\gamma_t, \gamma_{t+1}, \ldots$. Intuitively, $R(\gamma_t, v)$ has the following meaning: (i) when $\gamma_t(r_{v,p(v)}).\mathtt{est} \in \{1, 0^+\}$, v sends $\Delta(\gamma_t, v)$ tokens within $8(R(\gamma_t, v) + \Delta(\gamma_t, v))$ rounds in $\gamma_t, \gamma_{t+1}, \ldots$ if $p(v)$ always receives a token from $r_{v,p(v)}$ immediately after v sends a token to $r_{v,p(v)}$, and (ii) when $\gamma_t(r_{v,p(v)}).\mathtt{est} \in \{0^-, -1\}$, v receives $-\Delta(\gamma_t, v)$ tokens within $8(R(\gamma_t, v) - \Delta(\gamma_t, v))$ rounds in $\gamma_t, \gamma_{t+1}, \ldots$ if $p(v)$ always sends a token to $r_{p(v),v}$ immediately after $\mathit{IsReady}(p(v), v) \wedge r_{p(v),v}.\mathtt{est} = -1$ becomes true. For any process $v \in V$, we define $f(v)$ as follows:

$$f(v) = \begin{cases} R(v) + |\Delta(v)| + \left|\left\{u \in T_v \mid r_{u,p(u)}.\mathtt{est} \neq 0\right\}\right| & (v \neq v_{\mathrm{root}} \wedge r_{v,p(v)}.\mathtt{est} \neq \perp) \\ l + 3 + \sum_{u \in C(v)} f(u) & (v \neq v_{\mathrm{root}} \wedge r_{v,p(v)}.\mathtt{est} = \perp) \\ \sum_{u \in C(v)} f(u) & (v = v_{\mathrm{root}}). \end{cases}$$

▶ **Lemma 14.** *For any configuration γ, $f(\gamma, v_{\mathrm{root}}) = O(n\ell)$.*

Proof. Let v be any process other than v_{root}. Define $V' = \{u \in T_v \mid \gamma(r_{u,p(u)}).\mathtt{est} \neq \perp \wedge (u = v \vee \gamma(r_{p(u),p(p(u))}).\mathtt{est} = \perp)\}$. Then, $f(\gamma, v) \leq \sum_{u \in T_v}(\ell + 3) + \sum_{u \in V'}(R(\gamma, u) + |\Delta(\gamma, u)| + |\{u' \in T_u \mid \gamma(r_{u',p(u')}).\mathtt{est} \neq 0\}|) \leq n_v(\ell + 3) + \sum_{u \in V'}(h_u + n_u \cdot \ell + n_u) \leq n_v(\ell + 3) + \sum_{u \in V'} n_u(\ell + 2) \leq n_v(2\ell + 5)$. Therefore, $f(\gamma, v_{\mathrm{root}}) \leq \sum_{v \in C(v_{\mathrm{root}})} n_v(2\ell + 5) = O(n\ell)$. ◀

Note that $f(v)$ is monotonically non-increasing during any execution starting from a configuration in $\mathcal{C}_{\mathrm{est}}$ because both $R(v) + |\Delta(v)|$ and $|\{u \in T_v \mid r_{u,p(u)}.\mathtt{est} \neq 0\}|$ are monotonically non-increasing once $r_{v,p(v)} \neq \perp$ holds in such an execution. Moreover, detailed analysis gives Lemma 15. whose proof are omitted due to the lack of space.

▶ **Lemma 15.** *Let $\Gamma = \gamma_0, \gamma_1, \ldots$ be an execution starting from $\mathcal{C}_{\mathrm{est}}$. Then, $f(v_{\mathrm{root}})$ decrements at least by one in eight rounds of Γ as long as $f(v_{\mathrm{root}}) > 0$.*

▶ **Lemma 16.** *Every execution Γ of Base ends within $O(n\ell)$ rounds.*

Proof. Corollary 12, Lemma 14 and Lemma 15 imply that Γ reaches a configuration where $r_{v,p(v)}.\mathtt{est} = 0$ and $\Delta(v) = 0$ for all $v \in V \setminus \{v_{\mathrm{root}}\}$ within $O(n\ell)$ rounds. Pushing tokens to or pulling tokens from the external store, after an additional $O(\ell)$ rounds $\mathit{Diff}(v_{\mathrm{root}}) = 0$. ◀

▶ **Theorem 17.** *Algorithm Base is a silent and self-stabilizing token distribution algorithm, which uses no work space per process and only constant work space per register, converges in $O(n\ell)$ rounds, and causes $\Theta(nh\epsilon)$ redundant token moves.*

4.2 Algorithm *SyncTokenDist*

Due to the lack of space, we present only an outline of the algorithm *SyncTokenDist* which reduces the number of redundant token moves of *Base*. The key idea is simple. Corollary 12 and Lemma 13 guarantee that every execution of *Base* enters a configuration in $\mathcal{C}_{\text{est}}$ within $2h$ rounds and no redundant token moves happen thereafter. However, some processes can send or receive many tokens in the first $2h$ rounds, which makes $\Omega(nh\epsilon)$ redundant token moves in total in the worst case (Lemma 7). Algorithm *SyncTokenDist* simulates an execution of *Base* with a simplified version of the $\mathbb{Z}_3$ synchronizer [5], which loosely synchronizes an execution of *Base* so that the following property holds.

> For any integer x, if a process executes the procedure of *Base* at least $x+2$ times, then every neighboring process of the process must execute the procedure of *Base* at least x times.

This property and Lemma 10 guarantee that every process v can execute the procedure of *Base* at most $O(h)$ times until a configuration in $\mathcal{C}_{\text{est}}$ is reached, after which no redundant token moves happen.

Specifically, in *SyncTokenDist*, every process v has variable $v.\texttt{clock} \in \{0,1,2\}$ and $r_{v,u}.\texttt{clock} \in \{0,1,2\}$ for every $u \in N(v)$. It always copies the latest value of $v.\texttt{clock}$ to $r_{v,u}.\texttt{clock}$ for all $u \in N(v)$. It also has all the variables of *Base*. We say that a process u is ahead of v if $u.\texttt{clock} = v.\texttt{clock}+1 \pmod 3$. Process v increments its clock when it is ahead of no neighbors and it has a neighbor that is ahead of v or when v is enabled to execute the procedure of *Base* and $v.\texttt{clock} = u.\texttt{clock}$ holds for all neighbors $u \in N(v)$. Process v executes the procedure of *Base* every time v increments its clock. It is easy to see that this simple algorithm satisfies the above property, hence we obtain the following theorem.

▶ **Theorem 18.** *Algorithm SyncTokenDist is a silent and self-stabilizing token distribution algorithm, which uses only constant work space per process and per register, converges in $O(n\ell)$ rounds, and causes $O(nh)$ redundant token moves.*

4.3 Algorithm *PIFTokenDist*

In this section, we present *PIFTokenDist*, which reduces the number of redundant token moves of *Base* from $O(nh\epsilon)$ to $O(n)$ but increases the convergence time from $O(n\ell)$ to $O(nh\ell)$. Algorithm *PIFTokenDist* uses Propagation of Information with Feedback (PIF) scheme [3] to reduce the number of redundant token moves. For our purpose, we use a simplified version of PIF. The pseudo code is shown in Algorithm 2. Each process v has a local variable $v.\texttt{wave} \in \{0,1,2\}$, a shared variable $r_{v,u}.\texttt{wave} \in \{0,1,2\}$ for all $u \in N(v)$, and all the variables of *Base*. Process v always copies the latest value of $v.\texttt{wave}$ to $r_{v,u}.\texttt{wave}$ for all $u \in N(v)$ (Line 4). An execution of *PIFTokenDist* repeats the cycle of three waves – the 0-wave, the 1-wave, and the 2-wave (Figure 5). Once $v_{\text{root}}.\texttt{wave} = 0$, the zero value is propagated from v_{root} to leaves (Line 1, the 0-value). In parallel, each process v changes $v.\texttt{wave}$ from 0 to 1 after verifying that all its children already have the zero value in variable `wave` (Line 2, the 1-wave). When the 1-wave reaches a leaf, the wave bounces back to the root, changing the wave-value of processes from 1 to 2 (Line 3, the 2-wave). When the 2-wave reaches the root, it resets $v_{\text{root}}.\texttt{wave}$ to 0, thus the next cycle begins. A process v executes

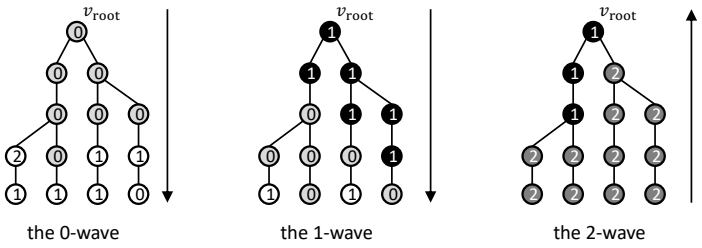

Figure 5 PIF waves of *PIFTokenDist*. A number in a circle represents the value of variable `wave`. Note that the 0-wave and the 1-wave run simultaneously.

Algorithm 2 *PIFTokenDist*.

[Actions of process v]

1: $v.\texttt{wave} \leftarrow 0$ **if** $(v = v_{\text{root}} \wedge v.\texttt{wave} = 2) \vee (v \neq v_{\text{root}} \wedge r_{p(v),v}.\texttt{wave} = 0)$
2: $v.\texttt{wave} \leftarrow 1$ **if** $(v.\texttt{wave} = 0) \wedge (v = v_{\text{root}} \vee r_{p(v),v}.\texttt{wave} = 1) \wedge \forall u \in C(v) : r_{u,v}.\texttt{wave} = 0)$
3: $v.\texttt{wave} \leftarrow 2$ and execute the procedure of *Base* **if** $(v.\texttt{wave} = 1) \wedge (\forall u \in C(v) : r_{u,v}.\texttt{wave} = 2)$
4: $r_{v,u}.\texttt{wave} \leftarrow v.\texttt{wave}$ for all $u \in N(v)$

the procedure of *Base* every time it receives the 2-wave, that is, every time it changes $v.\texttt{wave}$ from 1 to 2 (Line 3).

It is easy to see that every execution of *PIFTokenDist* reaches a configuration from which the cycle of three waves is repeated forever. Furthermore, the above PIF mechanism guarantees that, for any path $v_0, v_1, \ldots, v_\rho$ such that v_0 is a leaf, each process v_i $(1 \leq i \leq \rho)$ receives the 2-wave at least once in the order of $v_0, v_1, \ldots, v_\rho$ before v_ρ receives the 2-wave twice. Therefore, it holds by Lemma 10 that an execution of *PIFTokenDist* reaches a configuration in $\mathcal{C}_{\text{est}}$ before some process executes the procedure of *Base* more than three times. Hence, the number of redundant token moves is $O(n)$ in total. However, the convergence time increases from $O(n\ell)$ to $O(nh\ell)$ because it takes $O(h)$ rounds between every two consecutive executions of the procedure of *Base* at each process in an execution of *PIFTokenDist*. *PIFTokenDist*, shown in Algorithm 2 is not silent, but it can made silent by slightly modifying the algorithm, such that the root begins the 0-wave at Line 1 only when it detects that the simulated algorithm (*Base*) is not terminated. This modification is easily implemented by using the enabled-signal-propagation technique presented in [4].

▶ **Theorem 19.** *PIFTokenDist is a silent and self-stabilizing token distribution algorithm, which uses constant work space per process and per register, converges in $O(nh\ell)$ rounds, and permits $O(n)$ redundant token moves.*

5 Conclusion

We have given self-stabilizing and silent distributed algorithms for token distribution for rooted tree networks. The base algorithm *Base* converges in $O(n\ell)$ asynchronous rounds and causes $O(nh\epsilon)$ redundant token moves. Algorithms *SyncTokenDist* and *PIFTokenDist* use a synchronizer and a PIF scheme, respectively. Algorithm *SyncTokenDist* reduces the number of redundant token moves to $O(nh)$ without increasing convergence time while *PIFTokenDist* reduces the number of redundant token moves to $O(n)$, but increases the convergence time

to $O(nh\ell)$ rounds. All of the three algorithms uses constant memory space for each process and each link register.

References

1 Anish Arora and Mohamed G. Gouda. Load balancing: An exercise in constrainted convergence. In *Proceedings of the 9th International Workshop on Distributed Algorithms*, pages 183–197, 1995.

2 Andrei Z. Broder, Alan M. Frieze, Eli Shamir, and Eli Upfal. Near-perfect token distribution. In *Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, pages 308–317, 1992.

3 Alain Bui, Ajoy K Datta, Franck Petit, and Vincent Villain. Snap-stabilization and PIF in tree networks. *Distributed Computing*, 20(1):3–19, 2007.

4 Ajoy K Datta, Laurence L Larmore, Toshimitsu Masuzawa, and Yuichi Sudo. A self-stabilizing minimal k-grouping algorithm. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, pages 3:1–3:10. ACM, 2017.

5 Ajoy K. Datta, Lawrence L. Larmore, and Toshimitsu Masuzawa. Constant space self-stabilizing center finding in anonymous tree networks. In *Proceedings of the International Conference on Distributed Computing and Networking*, pages 38:1–38:10, 2015.

6 EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of Computing Machinery*, 17:643–644, 1974.

7 Bhaskar Ghosh, Frank Thomson Leighton, Bruce M. Maggs, S. Muthukrishnan, C. Greg Plaxton, Rajmohan Rajaraman, Andréa W. Richa, Robert Endre Tarjan, and David Zuckerman. Tight analyses of two local load balancing algorithms. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 548–558, 1995.

8 Kieran T. Herley. A note on the token distribution problem. *Inf. Process. Lett.*, 38(6):329–334, 1991.

9 Michael E. Houle, Antonios Symvonis, and David R. Wood. Dimension-exchange algorithms for load balancing on trees. In *Proceedings of the 9th International Colloquium on Structural Information and Communication Complexity*, pages 181–196, 2002.

10 Michael E. Houle, Ewan D. Tempero, and Gavin Turner. Optimal dimension-exchange token distribution on complete binary trees. *Theor. Comput. Sci.*, 220(2):363–376, 1999.

11 Michael E. Houle and Gavin Turner. Dimension-exchange token distribution on the mesh and the torus. *Parallel Computing*, 24(2):247–265, 1998.

12 Luciano Margara, Alessandro Pistocchi, and Marco Vassura. Perfect token distribution on trees. In *Proceedings of Structural Information and Communication Complexity, 11th International Colloquium*, pages 221–232, 2004.

13 D. Peleg. *Distributed computing: a locality-sensitive approach*, volume 5. Society for Industrial Mathematics, 2000.

14 David Peleg and Eli Upfal. The generalized packet routing problem. *Theor. Comput. Sci.*, 53:281–293, 1987.

15 David Peleg and Eli Upfal. The token distribution problem. *SIAM J. Comput.*, 18(2):229–243, 1989.

16 Yuichi Sudo, Ajoy K. Datta, Laurence L. Larmore, and Toshimitsu Masuzawa. Constant-space self-stabilizing token distribution in trees. In *Proceedings of 25th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 25–29, 2018.