# Self-Stabilizing Token Distribution with Constant-Space for Trees 

Yuichi Sudo<br>Osaka University, y-sudou@ist.osaka-u.ac.jp<br>Ajoy K. Datta<br>University of Nevada, Las Vegas, ajoy.datta@unlv.edu<br>Lawrence L. Larmore<br>University of Nevada, Las Vegas, lawrence.larmore@unlv.edu<br>Toshimitsu Masuzawa<br>Osaka University, masuzawa@ist.osaka-u.ac.jp

Follow this and additional works at: https://digitalscholarship.unlv.edu/compsci_fac_articles
Part of the Computer Sciences Commons

## Repository Citation

Sudo, Y., Datta, A. K., Larmore, L. L., Masuzawa, T. (2018). Self-Stabilizing Token Distribution with Constant-Space for Trees. 22nd International Conference on Principles of Distributed Systems (OPODIS 2018) 1-16. Hong Kong, China: Schloss Dagstuhl - Leibniz Center for Informatics.
http://dx.doi.org/10.4230/LIPIcs.OPODIS.2018.31

This Conference Proceeding is protected by copyright and/or related rights. It has been brought to you by Digital Scholarship@UNLV with permission from the rights-holder(s). You are free to use this Conference Proceeding in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself.

This Conference Proceeding has been accepted for inclusion in Computer Science Faculty Publications by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.

# Self-Stabilizing Token Distribution with Constant-Space for Trees 

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

Ajoy K. Datta<br>Department of Computer Science, University of Nevada, Las Vegas, USA<br>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 $\ell$ 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 $n k$ 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 $\Theta(n h \epsilon)$ 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 $O(n h)$ without any additional costs while the other reduces the number of redundant token moves to $O(n)$, but increases the convergence time to $O(n h \ell)$. All algorithms given have constant memory at each process and each link register.


2012 ACM Subject Classification Theory of computation $\rightarrow$ 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 $\ell$ 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
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 31; pp. 31:1-31:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
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 $n k$. Instead, each process holds an arbitrary number, from zero to $\ell$, 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; $\Theta(n h \epsilon)$ 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 $O(n h)$ 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 $O(n h \ell)$ 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 | $\Theta(n h \epsilon)$ | $(0, O(1))$ |
| SyncTokenDist | $O(n \ell)$ rounds | $O(n h)$ | $(O(1), O(1))$ |
| PIFTokenDist | $O(n h \ell)$ 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 $O(T d)$ 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 $v_{\text {root }}$ 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 $n_{v}=\left|T_{v}\right|$. 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 $\ell$ 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 $v$.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 $r_{u, v}$.token. The root process $v_{\text {root }}$ can also access the external token store, which contains infinitely many tokens. As we will see later, $v_{\text {root }}$ 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.tokenStore |), and two given parameters $\ell$ 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 $v \cdot \operatorname{Push}(x)$, it append a token with bit sequence $x$ into its token store; when it executes $v . \operatorname{Pull}()$, 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., $\mid v$. tokenStore $\mid=0$ ), and whether its local token store is full (i.e., $\mid v$. tokenStore $\mid=\ell$ ). 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 $v_{\text {root }}$ can push a token into or pull a token out of the external store by executing exPush $(x)$ and exPull(), respectively. Typically, it can move a token from its local token store to the external store by exPush(Pull()) and can move a token in the reverse direction by Push(exPull()). There is one restriction, however, each of exPush() and 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 $\gamma$ and $\gamma^{\prime}$ be two configurations of an algorithm $\mathcal{A}$ on $T$. We say that $\gamma \mapsto \gamma^{\prime}$ is a step of $\mathcal{A}$ if there is a non-empty set $S \subseteq V$ of enabled processes such that $\gamma$ changes to $\gamma^{\prime}$ 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 $\gamma \mapsto \gamma^{\prime}$. 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 $v . \operatorname{var} 1$, and shared variable var2 of register $r_{u, v}$ by $r_{u, v}$.var2. Furthermore, for any configuration $\gamma$, we denote the values of $v . \operatorname{var} 1$ and $r_{u, v}$.var2 in configuration $\gamma$ by $\gamma(v)$.var1 and $\gamma\left(r_{u, v}\right)$.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 $v_{\text {root }}$ pushes them into the external store, and that no new token appears in the network until $v_{\text {root }}$ pulls a token from

[^0]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 $\mathcal{P}_{\mathcal{A}}: R_{\mathcal{A}} \times R_{\mathcal{A}} \rightarrow\{$ 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 $\gamma$ 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 $\gamma \mapsto \gamma^{\prime}$ is legal with $\mathcal{P}_{\mathcal{A}}$ if the following three conditions hold: (i) process $v_{\text {root }}$ does not invoke both the two commands $\operatorname{exPush}(x)$ and $\operatorname{exPull}()$ in step $\gamma \mapsto \gamma^{\prime}$, (ii) if $v_{\text {root }}$ pulls a token $x$ from (resp. pushes a token $x$ to) the external store in step $\gamma \mapsto \gamma^{\prime}$, 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 $m_{\gamma^{\prime}, \mathcal{P}_{\mathcal{A}}}$, and (iii) if $v_{\text {root }}$ does not execute either exPush $(x)$ nor exPull() during step $\gamma \mapsto \gamma^{\prime}$, then $m_{\gamma, \mathcal{P}_{\mathcal{A}}}=m_{\gamma^{\prime}, \mathcal{P}_{\mathcal{A}}}$. 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 $\gamma \mapsto \gamma^{\prime}$ is a step of $\mathcal{A}$, then $\gamma^{\prime} \in \mathcal{L}_{\mathcal{A}}$. (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 $\gamma$ 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 $\gamma_{0}$. We define the first round of execution $\Gamma$ to be the smallest prefix, say $\gamma_{0}, \ldots, \gamma_{t}$, of $\Gamma$ 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 $\Gamma$. Let $\Gamma^{\prime}$ be the suffix of $\Gamma$ starting from $\gamma_{t}$. The second round of $\Gamma$ is defined to be the first round of $\Gamma^{\prime}$, and so forth.

It requires $l b$ 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 $\gamma \mapsto \gamma^{\prime}$ of algorithm $\mathcal{A}$ when predicate $\neg \mathcal{P}_{\mathcal{A}}\left(\gamma\left(r_{u, v}\right), \gamma\left(r_{v, u}\right)\right) \wedge \mathcal{P}_{\mathcal{A}}\left(\gamma^{\prime}\left(r_{u, v}\right), \gamma^{\prime}\left(r_{v, u}\right)\right)$ holds. Furthermore, introducing a virtual process $p\left(v_{\text {root }}\right) \notin V$, we also say that a token moves from $v_{\text {root }}$ to


Figure 1 A tree to prove Theorem 1.


Figure 2 A tree to prove Theorem 2.
$p\left(v_{\text {root }}\right)$ (resp. from $p\left(v_{\text {root }}\right)$ to $\left.v_{\text {root }}\right)$ in a step when $v_{\text {root }}$ invokes exPush() (resp. exPull()) in that step. We are interested in the number of redundant token moves. Given $v \in V$ and configuration $\gamma$, let $\tau(\gamma, v)$ be the number of tokens in input registers of $v$ in configuration $\gamma$. Then, we define $\Delta(\gamma, v)=\sum_{u \in T_{v}} d(\gamma, u)$ where $d(\gamma, u)=\mid \gamma(u)$.tokenStore $\mid+\tau(\gamma, u)-k$. 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 $\gamma$ 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 $\sum_{v \in V}\left|\Delta\left(\gamma_{0}, v\right)\right|$ 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 $\sum_{v \in V}\left|\Delta\left(\gamma_{0}, v\right)\right|$.

## 3 Lower Bounds

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

Proof. Consider a tree network of an even number of processes where $v_{\text {root }}$ has two children $u$ and $u^{\prime}$ where $n_{u}=n / 2$ and $n_{u^{\prime}}=n / 2-1$, and consider a configuration of $\mathcal{A}$ on the graph where $T_{u}$ holds $n_{u} \cdot l$ tokens in total and $T_{u^{\prime}}$ 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 $v_{\text {root }}$ and process $u^{\prime}$ must be selected by the scheduler $\Omega(n k)$ times to receive $n_{v} \cdot k$ tokens from $v_{\text {root }}$. Thus, there exists an execution of $\mathcal{A}$ starting from this configuration which takes $\max (\Omega(n(l-k)), \Omega(n k))=\Omega(n l)$ 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 $v_{\text {root }}$ has $n^{\prime}$ children $u_{1}, u_{2}, \ldots, u_{n^{\prime}}$, 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 $i=1,2, \ldots, n^{\prime}$, process $u_{i}$ holds $k+1$ tokens, $w_{i}$ holds no token, and $u_{i}, r_{u_{i}, v_{\text {root }}}, r_{v_{\text {root }}, u_{i}}, r_{u_{i}, w_{i}}$, and $r_{w_{i}, u_{i}}$ are in states such that $u_{i}$ sends a token to $v_{\text {root }}$ 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 $n^{\prime}=\lfloor n / 2\rfloor$ 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 $r_{u, v}$.exist and $r_{u, v}$.ready in addition to $r_{u, v}$.token. We use predicate $\mathcal{P}_{\text {Base }}(a, b) \equiv a$.exist $\wedge b$.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 $r_{u, v}$.exist $\wedge r_{v, u}$.ready holds. Let $u$ and $v$ be neighboring processes. We define four predicates used in the pseudocodes as follows:

$$
\begin{aligned}
\operatorname{IsReady}(u, v) & \equiv\left(\neg r_{u, v} \cdot \text { exist } \wedge r_{v, u} \cdot \text { ready }\right), \\
\operatorname{JustSent}(u, v) & \equiv\left(r_{u, v} \cdot \text { exist } \wedge r_{v, u} \text {.ready }\right), \\
\text { JustReceived }(u, v) & \equiv\left(r_{u, v} \text {.exist } \wedge \neg r_{v, u} \cdot \text { ready }\right), \\
\operatorname{NotReady}(u, v) & \equiv\left(\neg r_{u, v} \text {.exist } \wedge \neg r_{v, u} \cdot \text { ready }\right) .
\end{aligned}
$$

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 $\operatorname{IsReady}(u, v)$ and $\mid u$.tokenStore $\mid>0$, process $u$ may perform " $r_{u, v}$.token $\leftarrow$ Pull ()" and " $r_{u, v}$.exist $\leftarrow$ true". After that, $\operatorname{JustSent}(u, v)$ becomes true.
Rule 2 When JustSent $(u, v)$ and $\mid v$.tokenStore $\mid<l$, process $v$ may perform "Push $\left(r_{u, v}\right.$.token)" and " $r_{v, u}$.ready $\leftarrow$ false". After that, JustReceived $(u, v)$ becomes true.
Rule 3 When JustReceived $(u, v)$, process $u$ may perform " $r_{u, v}$.exist $\leftarrow$ false". After that, NotReady $(u, v)$ becomes true.
Rule 4 When $\operatorname{NotReady}(u, v)$, process $v$ may perform " $r_{v, u}$.ready $\leftarrow$ true". After that, IsReady $(u, v)$ becomes true.
By definition of $\mathcal{P}_{\text {Base }}$, a token exists in $r_{u, v}$ if and only if $\operatorname{JustSent}(u, v)$ is true. In any step $\gamma \mapsto \gamma^{\prime}$, none of the above rule changes $m_{\gamma, \mathcal{P}_{\text {Base }}}$, 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 $v_{\text {root }}$ pushes a token to and pulls a token from the external store. Specifically, $v_{\text {root }}$ pulls a token from the external store by Push(exPull()), and $v_{\text {root }}$ pushes a token to the external store by "exPush(Pull())". The former (resp. latter) operation is performed only when $\mid v_{\text {root }}$.tokenStore $\mid<l$ (resp. $\mid v_{\text {root }}$.tokenStore $\mid>0$ ). Shared variables token, exist, and ready are never updated in any way other than Rules 1-4.

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

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


### 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 $\Theta(n h \epsilon)$ redundant token moves.

Some functions take a configuration (e.g., $\gamma$ ) 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 $\operatorname{sgn}(x)$, that is, $\operatorname{sgn}(x)=1, \operatorname{sgn}(x)=0$, and $\operatorname{sgn}(x)=-1$ if $x>0, x=0$, and $x<0$, respectively.

```
Algorithm 1 Base
[Variables of process \(\boldsymbol{v}\) and register \(\boldsymbol{r}_{\boldsymbol{v}, u}\) ]
    \(v\). tokenStore \{an array containing at most \(l\) tokens \(\}\)
    \(r_{v, u}\) exist, \(r_{v, u}\).ready
    \(r_{v, u}\).est \(\in\left\{1,0^{+}, 0,0^{-},-1, \perp\right\}\left\{r_{v, u}\right.\).est \(\in\{1,0,-1\}\) if \(v\) is a leaf. \(\}\)
    \(r_{v, u}\).token \(\in 2^{b}\)
```


## [Actions of process $\boldsymbol{v}$ ]

```
    \(r_{v, u}\).exist \(\leftarrow\) false for all \(u \in N(v)\) such that \(\operatorname{JustReceived}(v, u)\)
    \(r_{v, u}\).ready \(\leftarrow\) true for all \(u \in N(v)\) such that \(\operatorname{NotReady}(u, v)\)
    \(\operatorname{ReceiveToken}(u)\) for all \(u \in N(v)\) such that \(\operatorname{JustSent}(u, v)\)
    SendToken \((u)\) for all \(u \in C(v)\) such that \(r_{u, v}\).est \(=-1 \wedge \operatorname{IsReady}(v, u)\)
    if \(v=v_{\text {root }}\) then
        AdjustTokens()
    else
        \(\operatorname{SendToken}(p(v))\) if \(\operatorname{Est}(v)=1 \wedge \operatorname{IsReady}(v, p(v))\)
        \(r_{v, p(v)}\).est \(\leftarrow \operatorname{Est}(v)\)
    end if
    SendToken \((u)]\) :
    if \(\mid v\).tokenStore \(\mid>0\) then
        \(r_{v, u}\).token \(\leftarrow \operatorname{Pull}()\)
        \(r_{v, u}\).exist \(\leftarrow\) true
        \(\operatorname{ReceiveToken}(w)\) if \(\exists w \in N(v): \operatorname{JustSent}(w, v)\)
    end if
[ReceiveToken \((u)\) ]:
    6: if \(\mid v\).tokenStore \(\mid<l\) then
        Push \(\left(r_{u, v}\right.\).token \()\)
        \(r_{v, u}\).ready \(\leftarrow\) false
    end if
[AdjustTokens()]:
    if \(\operatorname{Est}(v)=-1\) then
    Push(exPull())
    else if \(\operatorname{Est}(v)=1\) then
        exPush(Pull())
        ReceiveToken \((u)\) if \(\exists u \in N(v): \operatorname{JustSent}(u, v)\)
    end if
```

During an execution of Base, each process $v$ tries to estimate $\operatorname{sgn}(\Delta(v))$, 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 $\operatorname{IsReady}(p(v), v)$. When the estimate is positive, $v$ sends a token to its parent $p(v)$ if $v$ holds a token and $\operatorname{IsReady}(v, p(v))$. The root $v_{\text {root }}$ always pulls a new token from the external store to increase $\Delta\left(v_{\text {root }}\right)$ when its estimate is negative, and pushes a token to the external store to decrease $\Delta\left(v_{\text {root }}\right)$ when the estimate is positive. If all processes $v$ correctly estimate $\operatorname{sgn}(\Delta(v))$, each of them eventually holds $k$ tokens. After that, no process sends a token.

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

$$
\begin{aligned}
& \operatorname{Est}(v)=\left\{\begin{array}{rr}
1 \begin{array}{rr}
1 & \left(\operatorname{Diff}(v)>0 \wedge \forall u \in C(v): r_{u, v} \cdot \text { est } \in\left\{1,0^{+}, 0\right\}\right) \\
0^{+} & \left(\operatorname{Diff}(v)=0 \wedge \forall u \in C(v): r_{u, v} \cdot \text { est } \in\left\{1,0^{+}, 0\right\}\right. \\
0 & \left(\operatorname{Diff}(v)=0 \wedge \forall u \in C(v): r_{u, v} \cdot \text { est }=0\right) \\
0^{-} & \left(\operatorname{Diff}(v)=0 \wedge \forall u \in C(v): r_{u, v} \cdot \text { est } \in\left\{-1,0^{-}, 0\right\}\right. \\
-1 & \left(\text { Diff }(v)<0 \wedge \forall u \in C(v): r_{u, v} \cdot \text { est } \in\left\{-1,0^{-}, 0\right\}\right) \\
\perp & \text { (otherwise) },
\end{array} \\
\operatorname{Diff(v)}=\mid v . \text {.tokenStore }|+|\{u \in N(v) \mid \operatorname{JustSent}(u, v)\}|-k
\end{array}\right.
\end{aligned}
$$

where the candidate values $1,0^{+}, 0,0^{-},-1$, and $\perp$ of $\operatorname{Est}(v)$ 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 $v \in V \backslash\left\{v_{\text {root }}\right\}$, the domain of variable $r_{v, p(v)}$.est is $\left\{1,0^{+}, 0,0^{-},-1\right\}$ 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 $r_{v, u}$.exist and $r_{v, u}$.ready for all $u \in N(v)$ such that $\operatorname{JustReceived}(v, u)$ or $\operatorname{NotReady}(u, v)$, 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 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 $\operatorname{IsReady}(v, u)$ and $r_{u, v}$.est $=-1$ (i.e., $\Delta(u)$ is estimated to be negative) by invoking $\operatorname{SendToken}(u)$ if $v$ 's token store is not empty (Lines 4, 11-15). Next, $v \in V \backslash\left\{v_{\text {root }}\right\}$ and $v_{\text {root }}$ perform different action. If $v \neq v_{\text {root }}$, it sends a token to its parent if $\operatorname{IsReady}(v, p(v))$ and $\operatorname{Est}(v)=1$, i.e., $\Delta(v)$ is estimated to be positive, and reports the latest estimate of $\operatorname{sgn}(\Delta(v))$ to its parent, and stores $E s t(v)$ into $r_{v, p(v)}$.est (Lines 7-10 ). Note that $\operatorname{Diff}(v)$ decreases when $v$ sends a token to its parent, hence the value of $\operatorname{Est}(v)$ may differ in Lines 8 and 9. Process $v_{\text {root }}$ invokes AdjustTokens() by which $v_{\text {root }}$ 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 $\operatorname{Est}(v)$ and $\operatorname{Diff}(v)$. 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 $\operatorname{Est}(v)$ in configuration $\gamma$ by $\operatorname{Est}(\gamma, v)$ and denote $\operatorname{Diff}(v)$ in configuration $\gamma$ by $\operatorname{Diff}(\gamma, v)$.

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 $P_{\text {finite }}(v) \equiv P_{\text {finiteMove }}(v) \wedge P_{\text {finiteEst }}(v)$, where $P_{\text {finiteMove }}(v)$ and $P_{\text {finiteEst }}(v)$ are also predicates: $P_{\text {finiteMove }}(v)$ holds if and only if $v$ sends and receives tokens only finitely many times in $\Gamma$, and $P_{\text {finiteEst }}(v)$ holds if and only if $v=v_{\text {root }}$ or $v$ changes the value
of $r_{v, p(v)}$.est only finitely many times in $\Gamma$. In the remainder of this proof, we prove $\left(\forall u \in C(v) ; P_{\text {finite }}(u)\right) \Rightarrow P_{\text {finite }}(v)$ 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, $P_{\text {finite }}\left(v^{\prime}\right)$ 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 $r_{u, v}$. 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 $\operatorname{Diff}(v)>0$, and $p(v)$ sends a token to $v$ only if Diff $(v)<0$. Hence, $v$ sends and receives tokens at most $\mid$ Diff $\left(\gamma_{t+1}, v\right) \mid$ times in $\Gamma_{t+1}$, which implies $P_{\text {finiteMove }}(v)$. Thus, there exists $t^{\prime}>t$ such that $v$ never sends or receives a token after $\gamma_{t^{\prime}}$. Since $\operatorname{Diff}(v)$ never changes after $\gamma_{t^{\prime}}, \operatorname{Est}(v)$ also never changes after $\gamma_{t^{\prime}}$. This means that $r_{v, p(v)}$.est never changes after $\gamma_{t^{\prime}+1}$, 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 $\gamma$ ends at its final configuration $\gamma_{f}$, that is, $\Gamma=\gamma_{0}, \gamma_{1}, \ldots, \gamma_{f}$. No process is enabled in $\gamma_{f}$. Hence, every process holds $k$ tokens in its token store and no token exists in registers in $\gamma_{f}$ 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 $O(n h \epsilon)$.

Proof. For any process $v$, let $\Delta^{+}(v)=\sum_{u \in T_{v}} \max (\operatorname{Diff}(u), 0)$ and $\Delta^{-}(v)=\sum_{u \in T_{v}}$ $\max (-$ Diff $(u), 0)$. During the execution $\Gamma=\gamma_{0}, \gamma_{1}, \ldots$ of Base, each $v$ sends a token to $p(v)$ at most $\Delta^{+}\left(\gamma_{0}, v\right)+n_{v}$ 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 $\Delta^{-}\left(\gamma_{0}, v\right)$ times in $\Gamma$. Since $\Delta\left(\gamma_{0}, v\right)=\Delta^{+}\left(\gamma_{0}, v\right)-\Delta^{-}\left(\gamma_{0}, v\right)$ and $\sum_{v \in V} n_{v} \leq n h$, the number of redundant token moves in $\Gamma$ is at most $\sum_{v \in V}\left(\Delta^{+}\left(\gamma_{0}, v\right)+\Delta^{-}\left(\gamma_{0}, v\right)+n_{v}\right)-\sum_{v \in V}\left|\Delta\left(\gamma_{0}, v\right)\right|=$ $\sum_{v \in V}\left(2 \min \left(\Delta^{+}\left(\gamma_{0}, v\right), \Delta^{-}\left(\gamma_{0}, v\right)\right)+n_{v}\right) \leq \sum_{v \in V}\left(2 \min \left(n_{v}(l-k), n_{v} \cdot k\right)+n_{v}\right)=O(n h \epsilon)$.

- Lemma 7. The number of redundant token moves in an execution of Base is $\Omega(n h \epsilon)$.

Proof. Consider the network shown in Figure 3 where $2 x \cdot 2 y$ processes except for $v_{\text {root }}$ are divided into four sets $\mathrm{A}, \mathrm{B}, \mathrm{C}$, and D , each of which consists of $x y$ processes. Consider a configuration where each process $v$ in $\mathrm{A} \cup \mathrm{D}$ holds no token (i.e., Diff $(v)=-k$ ), each process $u$ in $\mathrm{B} \cup \mathrm{C}$ holds $\ell$ tokens (i.e., Diff $(u)=\ell-k$ ), and $r_{w_{i}, p\left(w_{i}\right)}$. est $=r_{w_{i}^{\prime}, p\left(w_{i}^{\prime}\right)}=0$ holds for all $i=1,2, \ldots, y$. In an execution starting from the configuration, the scheduler can select processes only in $\mathrm{A} \cup \mathrm{D} \cup\left\{v_{\text {root }}\right\}$ until all processes in $\mathrm{A} \cup \mathrm{D} \cup\left\{v_{\text {root }}\right\}$ become disabled. Such an execution must make at least $\min (x(x-1) y k / 2, x(x-1) y(l-k) / 2)=\Omega(n h \epsilon)$ 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 $r_{v, p(v)}$.est. Specifically, we prove that every execution of Base satisfies predicate $P_{\text {est }}(v)$, which is defined as follows, within $2 h_{v}$ rounds, for any process $v$ other than $v_{\text {root }}$ :

$$
\begin{aligned}
P_{\mathrm{est}}(v) & \equiv P_{1}(v) \wedge P_{2}(v) \wedge P_{3}(v) \wedge P_{4}(v) \wedge Q_{1}(v) \wedge Q_{2}(v) \wedge Q_{3}(v) \wedge \forall u \in C(v): P_{\mathrm{est}}(u) \\
P_{1}(v) & \equiv r_{v, p(v)} \cdot \mathrm{est} \in\left\{1,0^{+}\right\} \Rightarrow \Delta(v)>0 \wedge D i f f(v) \geq 0 \\
P_{2}(v) & \equiv r_{v, p(v)} \cdot \mathrm{est}=0 \Rightarrow \Delta(v)=0 \wedge D i f f(v)=0 \\
P_{3}(v) & \equiv r_{v, p(v) \cdot \mathrm{est}}=0^{-} \Rightarrow \Delta(v) \leq 0 \wedge \operatorname{Diff}(v) \leq 0 \\
P_{4}(v) & \equiv r_{v, p(v) \cdot \mathrm{est}}=-1 \Rightarrow(\Delta(v)<0 \vee(\Delta(v)=0 \wedge \operatorname{JustSent}(p(v), v))) \\
& \wedge \operatorname{Diff}(v) \leq 0 \\
Q_{1}(v) & \equiv r_{v, p(v) \cdot \mathrm{est} \in\left\{1,0^{+}\right\} \Rightarrow \forall u \in T_{v}: r_{u, p(u)} \cdot \mathrm{est} \in\left\{1,0^{+}, 0\right\}} \\
Q_{2}(v) & \equiv r_{v, p(v) \cdot \mathrm{est}}=0 \Rightarrow \forall u \in T_{v}: r_{u, p(u)} \cdot \mathrm{est}=0, \\
Q_{3}(v) & \equiv r_{v, p(v)} \cdot \mathrm{est} \in\left\{0^{-},-1\right\} \Rightarrow \forall u \in T_{v}: r_{u, p(u)} \cdot \mathrm{est} \in\left\{0,0^{-},-1\right\} .
\end{aligned}
$$

- Lemma 8. Let $v \in V \backslash\left\{v_{\text {root }}\right\}$ and let $\gamma \mapsto \gamma^{\prime}$ be a step of Base where $P_{\text {est }}(\gamma, v)$ holds. Then, the following three statements hold:

$$
\begin{align*}
\gamma\left(r_{v, p(v)}\right) \text {.est }=0 & \Rightarrow \gamma^{\prime}\left(r_{v, p(v)}\right) \text {.est }=0  \tag{1}\\
\gamma\left(r_{v, p(v)}\right) \text {.est } \in\left\{1,0^{+}\right\} & \Rightarrow \gamma^{\prime}\left(r_{v, p(v)}\right) \text {.est } \in\left\{1,0^{+}, 0\right\}  \tag{2}\\
\gamma\left(r_{v, p(v)}\right) \text {.est } \in\left\{0^{-},-1\right\} & \Rightarrow \gamma^{\prime}\left(r_{v, p(v)}\right) . \text { est } \in\left\{0,0^{-},-1\right\} \tag{3}
\end{align*}
$$

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

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

- Lemma 10. Let $v \in V$. If $P_{\text {est }}(u)$ holds for all $u \in C(v), P_{\text {est }}(v)$ 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 $P_{\text {est }}\left(\gamma_{t}, u\right)$ holds for all $t \geq 0$ and all $u \in C(v)$. During the execution, $P_{\text {est }}(v)$ holds when $v$ is disabled, by the definition of algorithm Base, and by the assumption that $P_{\text {est }}(u)$ holds for all $u \in C(v)$. Hence, it suffices to show that $P_{\text {est }}\left(\gamma_{t+1}, v\right)$ or $P_{\text {est }}\left(\gamma_{t^{\prime}+1}, v\right)$ holds where $\gamma_{t} \mapsto \gamma_{t+1}$ and $\gamma_{t^{\prime}} \mapsto \gamma_{t^{\prime}+1}$ are the first and the second steps where $v$ is selected by the scheduler in $\Gamma$. Due to the token passing mechanism, $p(v)$ cannot send a token to $r_{p(v), v}$ in both steps. For any step $\gamma \mapsto \gamma^{\prime}$, $P_{\text {est }}\left(\gamma^{\prime}, v\right)$ holds by the definition of Base if $v$ is selected by the scheduler in $\gamma \mapsto \gamma^{\prime}, p(v)$ cannot send a token to $r_{p(v), v}$ at the step, and $P_{\text {est }}(\gamma, u) \wedge P_{\text {est }}\left(\gamma^{\prime}, u\right)$ holds for all $u \in C(v)$. Hence, $P_{\text {est }}\left(\gamma_{t+1}, v\right)$ or $P_{\text {est }}\left(\gamma_{t^{\prime}+1}, v\right)$ holds.

- Lemma 11. Let $v \in V$ and let $\Gamma=\gamma_{0}, \gamma_{1}, \ldots$ be an execution of Base. The predicate $P_{\text {est }}\left(\gamma_{t}, v\right)$ holds for any $t \geq s\left(\Gamma, 2 h_{v}\right)$, i.e., $P_{\text {est }}(v)$ always holds after $2 h_{v}$ rounds have elapsed.

Proof. By induction on $h_{v}$.
We define $\mathcal{C}_{\text {est }}$ to be the set of all configurations where $P_{\text {est }}(v)$ holds for all $v \neq v_{\text {root }}$. 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 $\mathcal{C}_{\text {est }}$.

- Corollary 12. An execution of Base reaches a configuration of $\mathcal{C}_{\text {est }}$ within $2 h$ rounds and never deviates from $\mathcal{C}_{\text {est }}$ thereafter.
- Lemma 13. For any process $v,|\Delta(v)|$ is monotonically non-increasing during an execution of Base starting from a configuration in $\mathcal{C}_{\text {est }}$.

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

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

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 $\left.C_{1}(v)=\{v\} \cup C(v)\right)$. See Figure 4. In the left tree of the figure, $R(v)=1$ because $C_{2 j+1}(v)$ holds at least $j$ extra tokens (i.e., $\sum_{u \in C_{2 j+1}(v)}$ 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 $C_{2 j+5}(v)$ is short at least $j$ tokens (i.e., $\sum_{u \in C_{2 j+5}(v)}(-\operatorname{Diff}(v)) \geq j$ ) 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 Diff $(w)$, omitted when $\operatorname{Diff}(w)=0$.
configuration in $\mathcal{C}_{\text {est }}$ and $v \in V \backslash\left\{v_{\text {root }}\right\}$ be a process such that $r_{v, p(v)}$. est $\neq \perp$ holds for some configuration $\gamma_{t}$. Then either $\forall u \in T_{v}: r_{u, p(u)}$. est $\in\left\{1,0^{+}, 0\right\}$ or $\forall u \in T_{v}: r_{u, p(u)}$. est $\in$ $\left\{0,0^{-},-1\right\}$ holds for $\gamma_{t}, \gamma_{t+1}, \ldots$. Intuitively, $R\left(\gamma_{t}, v\right)$ has the following meaning: (i) when $\gamma_{t}\left(r_{v, p(v)}\right)$.est $\in\left\{1,0^{+}\right\}, v$ sends $\Delta\left(\gamma_{t}, v\right)$ tokens within $8\left(R\left(\gamma_{t}, v\right)+\Delta\left(\gamma_{t}, v\right)\right)$ 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 $\gamma_{t}\left(r_{v, p(v)}\right)$.est $\in\left\{0^{-},-1\right\}$, $v$ receives $-\Delta\left(\gamma_{t}, v\right)$ tokens within $8\left(R\left(\gamma_{t}, v\right)-\Delta\left(\gamma_{t}, v\right)\right)$ rounds in $\gamma_{t}, \gamma_{t+1}, \ldots$ if $p(v)$ always sends a token to $r_{p(v), v}$ immediately after $\operatorname{IsReady}(p(v), v) \wedge r_{p(v), v}$.est $=-1$ becomes true. For any process $v \in V$, we define $f(v)$ as follows:

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

- Lemma 14. For any configuration $\gamma, f\left(\gamma, v_{\text {root }}\right)=O(n \ell)$.

Proof. Let $v$ be any process other than $v_{\text {root }}$. Define $V^{\prime}=\left\{u \in T_{v} \mid \gamma\left(r_{u, p(u)}\right)\right.$.est $\neq \perp \wedge(u=$ $v \vee \gamma\left(r_{p(u), p(p(u))}\right)$.est $\left.\left.=\perp\right)\right\}$. Then, $f(\gamma, v) \leq \sum_{u \in T_{v}}(\ell+3)+\sum_{u \in V^{\prime}}(R(\gamma, u)+|\Delta(\gamma, u)|+$ $\mid\left\{u^{\prime} \in T_{u} \mid \gamma\left(r_{u^{\prime}, p\left(u^{\prime}\right)}\right)\right.$.est $\left.\left.\neq 0\right\} \mid\right) \leq n_{v}(\ell+3)+\sum_{u \in V^{\prime}}\left(h_{u}+n_{u} \cdot \ell+n_{u}\right) \leq n_{v}(\ell+3)+$ $\sum_{u \in V^{\prime}} n_{u}(\ell+2) \leq n_{v}(2 \ell+5)$. Therefore, $f\left(\gamma, v_{\text {root }}\right) \leq \sum_{v \in C\left(v_{\text {root }}\right)} n_{v}(2 \ell+5)=O(n \ell)$.

Note that $f(v)$ is monotonically non-increasing during any execution starting from a configuration in $\mathcal{C}_{\text {est }}$ because both $R(v)+|\Delta(v)|$ and $\mid\left\{u \in T_{v} \mid r_{u, p(u)}\right.$.est $\left.\neq 0\right\} \mid$ 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 $\mathcal{C}_{\text {est }}$. Then, $f\left(v_{\text {root }}\right)$ decrements at least by one in eight rounds of $\Gamma$ as long as $f\left(v_{\text {root }}\right)>0$.
- Lemma 16. Every execution $\Gamma$ of Base ends within $O(n \ell)$ rounds.

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

- 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 $\Theta(n h \epsilon)$ 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 $2 h$ rounds and no redundant token moves happen thereafter. However, some processes can send or receive many tokens in the first $2 h$ rounds, which makes $\Omega(n h \epsilon)$ 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 $v . c l o c k \in\{0,1,2\}$ and $r_{v, u} . c l o c k \in\{0,1,2\}$ for every $u \in N(v)$. It always copies the latest value of $v . c l o c k ~ t o$ $r_{v, u}$.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 $u . c l o c k=v . c l o c k+1(\bmod 3)$. 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 $v . c l o c k=u . c l o c k ~ h o l d s$ 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 $O(n h)$ redundant token moves.


### 4.3 Algorithm PIFTokenDist

In this section, we present PIFTokenDist, which reduces the number of redundant token moves of Base from $O(n h \epsilon)$ to $O(n)$ but increases the convergence time from $O(n \ell)$ to $O(n h \ell)$. 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 $v$.wave $\in\{0,1,2\}$, a shared variable $r_{v, u}$.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 $v$.wave to $r_{v, u}$.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 $v_{\text {root }}$.wave $=0$, the zero value is propagated from $v_{\text {root }}$ to leaves (Line 1 , the 0 -value). In parallel, each process $v$ changes $v$.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 $v_{\text {root }}$ 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 \(\boldsymbol{v}\) ]
    \(v\).wave \(\leftarrow 0\) if \(\left(v=v_{\text {root }} \wedge v\right.\).wave \(\left.=2\right) \vee\left(v \neq v_{\text {root }} \wedge r_{p(v), v}\right.\).wave \(\left.=0\right)\)
    \(v\).wave \(\leftarrow 1\) if \((v\).wave \(=0) \wedge\left(v=v_{\text {root }} \vee r_{p(v), v}\right.\).wave \(\left.=1\right) \wedge \forall u \in C(v): r_{u, v}\).wave \(\left.=0\right)\)
    \(v\).wave \(\leftarrow 2\) and execute the procedure of Base if \((v\). wave \(=1) \wedge\left(\forall u \in C(v): r_{u, v}\right.\).wave \(=\)
    2)
4: \(r_{v, u \text {.wave }}^{\leftarrow v \text {.wave for all } u \in N(v), ~(x)}\)
```

the procedure of Base every time it receives the 2 -wave, that is, every time it changes $v$.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 $v_{\rho}$ 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 $O(n h \ell)$ 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 $O(n h \ell)$ 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 $O(n h \epsilon)$ 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 $O(n h)$ without increasing convergence time while PIFTokenDist reduces the number of redundant token moves to $O(n)$, but increases the convergence time
to $O(n h \ell)$ 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 selfstabilizing 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 selfstabilizing 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):329334, 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):229243, 1989.
16 Yuichi Sudo, Ajoy K. Datta, Laurence L. Larmore, and Toshimitsu Masuzawa. Constantspace self-stabilizing token distribution in trees. In Proceedings of 25th International Colloquium on Structural Information and Communication Complexity (SIROCCO), pages 25-29, 2018.


[^0]:    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.

