Constant-Space Self-stabilizing Token Distribution in Trees

Yuichi Sudo[1(✉)], Ajoy K. Datta[2], Lawrence L. Larmore[2], and Toshimitsu Masuzawa[1]

[1] Osaka University, 1-5, Yamadaoka, Suita, Osaka, Japan
y-sudou@ist.osaka-u.ac.jp

[2] University of Nevada, Las Vegas, 4505 S Maryland Pkwy, Las Vegas, NV, USA

1 Introduction

The token distribution problem was originally defined by Peleg and Upfal in their seminal paper [4]. 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 uniformly distribute the tokens such that every process ends up with exactly one token. We generalize this problem as follows: the goal is to distribute nk tokens such that every process holds k tokens where k is any given number. We present a self-stabilizing algorithm that solves this generalized problem. 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. Each process holds an arbitrary number, from zero to l, of tokens in an initial configuration. Thus, 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, $O(nl)$ (asynchronous) rounds. However, *Base* may have a large number of redundant token moves; $\Theta(nh\epsilon)$ *redundant* (or unnecessary) token moves happen in the worst case where $\epsilon = \min(k, l-k)$ where h is the height of the tree network. Next, we combine the algorithm *Base* with a synchronizer or 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 any additional costs while *PIFTokenDist* drastically reduces the number of redundant token moves to the asymptotically optimal value, $O(n)$, at the expense of increasing convergence time from $O(nl)$ to $O(nhl)$ in terms of rounds. Work space complexities, i.e., the amount of memory to store information except for tokens, of all the algorithms are constant both per process and per link register.

This work was partially supported by Japan Science and Technology Agency (JST) SICORP.

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

	Conv. time	#Red. token moves	Work space (Process)	Work space (Link)
Base	$O(nl)$ rounds	$\Theta(nh\epsilon)$	0	$O(1)$
SyncTokenDist	$O(nl)$ rounds	$O(nh)$	$O(1)$	$O(1)$
PIFTokenDist	$O(nhl)$ rounds	$O(n)$	$O(1)$	$O(1)$
Lowerbounds	$\Omega(nl)$ rounds	$\Omega(n)$	-	-

2 Preliminaries

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. The tree network is rooted, that is, there exists a designated process $v_{\text{root}} \in V$, and every process v other than v_{root} knows its parent $p(v)$. We denote the set of v's children by $C(v)$. We define $N(v) = C(v) \cup \{p(v)\}$. Each link $\{u, v\} \in E$ has two *link registers* or just *registers* $r_{u,v}$ and $r_{v,u}$. We call $r_{u,v}$ (resp. $r_{v,u}$) an output register (resp. an input register) of u. Process u can read from the both registers and can write only to output register $r_{u,v}$.

A process v holds at most l tokens at a time, each of which is a bit sequence of length b. These tokens are stored in a dedicated memory space of the process, called *token store*. We denote the token store of v by $v.\texttt{tokenStore}$ and the number of the tokens in it by $|v.\texttt{tokenStore}|$. 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 access another token store called *the external token store*, in which an infinite number of tokens exist. The root v_{root} can reduce the total number of tokens in the tree by pushing a token into the external store and can increase it by pulling a token from the external store.

Given $k \leq l$, our goal is to reach a configuration where every process holds exactly k tokens in a self-stabilizing fashion. All tokens must not disappear from the network except in the case that root v_r pushes them to the external store. A process must not create a new token. A new token appears only when the root pulls it from the external store.

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*. We measure the space complexity as the *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`). We evaluate the number of token moves as follows. Generally, a token is transferred from a process u to a process v in the following two steps: (i) u moves the token from $u.\texttt{tokenStore}$ to $r_{u,v}.\texttt{token}$; (ii) v moves the token from $r_{u,v}.\texttt{token}$ to $v.\texttt{tokenStore}$. 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. We are interested in *the number of redundant token moves.* Let $\tau(v)$ be the number of tokens in input registers of process v. Then, we define $\mathit{\Delta}(v) = \sum_{u \in T_v} d(u)$ where $d(u) = |u.\texttt{tokenStore}| + \tau(u) - k$ and T_v is the sub-tree consisting of all the descendants of v (including v itself). Intuitively, $\mathit{\Delta}(v)$ is the number of tokens that v must send to $p(v)$ to achieve the token distribution if $\mathit{\Delta}(v) \geq 0$; Otherwise, $p(v)$ must send $-\mathit{\Delta}(v)$ tokens to v. We define the number of redundant token moves in an execution as the total number of token moves in the execution minus $\sum_{v \in V} |\mathit{\Delta}(v)|$ of the initial configuration of the execution.

3 Algorithms

3.1 Algorithm *Base*

We use common notation $\mathrm{sgn}(x)$ for real number 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.

The basic idea of *Base* is simple. Each process v always tries to estimate $\mathrm{sgn}(\mathit{\Delta}(v))$, that is, tries to find whether $\mathit{\Delta}(v)$ is positive, negative, or just zero. Then, process v other than v_{root} reports its estimation to its parent $p(v)$ using a shared variable $r_{v,p(v)}.\texttt{est}$. When its estimation is negative, $p(v)$ sends a token to v if $p(v)$ holds a token and $r_{p(v),v}.\texttt{token}$ is empty. When the estimation is positive, v sends a token to its parent $p(v)$ if v holds a token and $r_{v,p(v)}.\texttt{token}$ is empty. Root v_{root} always pulls a new token from the external store to increase $\mathit{\Delta}(v_{\mathrm{root}})$ when its estimation is negative, and pushes a token to the external store to decrease $\mathit{\Delta}(v_{\mathrm{root}})$ when the estimation is positive. If all processes v correctly estimate $\mathrm{sgn}(\mathit{\Delta}(v))$, each of them eventually holds k tokens. After that, no process sends a token.

Thus, estimating $\mathrm{sgn}(\mathit{\Delta}(v))$ is the key of algorithm *Base*. Each process v estimates $\mathrm{sgn}(\mathit{\Delta}(v))$ as follows (*Est*(v) is the estimation):

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

where the candidate values 1, 0^+, 0, 0^-, -1, and $\bot$ of *Est*(v) represent that the estimation is positive, "never negative", zero, "never positive", negative, and "unsure", respectively. A process sends a token to its parent only when its estimation is 1, and it send a token to its child only when the estimation of the child is -1. Our detailed analysis proves that this simple constant-space algorithm shows the performance listed in Table 1.

3.2 Algorithm *SyncTokenDist*

The key idea of *SyncTokenDist* is simple. It is guaranteed that every process v has correct estimation in variable `est` within $2h$ asynchronous rounds and no redundant token moves happen thereafter. However, some processes can send or receive many tokens in the first $2h$ asynchronous rounds, which makes $\Omega(nh\epsilon)$ redundant token moves in total in the worst case. Algorithm *SyncTokenDist* simulates an execution of *Base* with a simplified version of the $\mathbb{Z}_3$ synchronizer [3], 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.

Thus, every process v can execute the procedure of *Base* at most $O(h)$ times until all agents have correct estimation, after which no redundant token moves happen.

3.3 Algorithm *PIFTokenDist*

Algorithm *PIFTokenDist* uses Propagation and Information with Feedback (PIF) scheme [1] 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 1. Each process v has a local variable $v.\mathtt{wave} \in \{0,1,2\}$, a shared variable $r_{v,u}.\mathtt{wave} \in \{0,1,2\}$ for all $u \in N(v)$, and all the variables of *Base*. Process v always copies the latest value of $v.\mathtt{wave}$ to $r_{v,u}.\mathtt{wave}$ 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. Once $v_{\mathrm{root}}.\mathtt{wave} = 0$, the zero value is propagated from v_{root} to leaves (Line 1, the 0-value). In parallel, each process v changes $v.\mathtt{wave}$ 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 $v_{\mathrm{root}}.\mathtt{wave}$ to 0, thus the next cycle begins. A process v executes the procedure of *Base* every time it receives the 2-wave, that is, every time it changes $v.\mathtt{wave}$ from 1 to 2 (Line 3).

Algorithm 1 *PIFTokenDist*

```
[Actions of process $v$]
1: $v.\mathtt{wave} \leftarrow 0$ if $(v = v_{\mathrm{root}} \land v.\mathtt{wave} = 2) \lor (v \neq v_{\mathrm{root}} \land r_{p(v),v}.\mathtt{wave} = 0)$
2: $v.\mathtt{wave} \leftarrow 1$ if $(v.\mathtt{wave} = 0) \land (v = v_{\mathrm{root}} \lor r_{p(v),v}.\mathtt{wave} = 1) \land \forall u \in C(v) : r_{u,v}.\mathtt{wave} = 0)$
3: $v.\mathtt{wave} \leftarrow 2$ and execute the procedure of Base if $(v.\mathtt{wave} = 1) \land (\forall u \in C(v) : r_{u,v}.\mathtt{wave} = 2)$
4: $r_{v,u}.\mathtt{wave} \leftarrow v.\mathtt{wave}$ for all $u \in N(v)$
```

The *PIFTokenDist* shown in Algorithm 1 is not silent, but it can get the silence property with slight modification 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 [2].

References

1. Bui, A., Datta, A.K., Petit, F., Villain, V.: Snap-stabilization and PIF in tree networks. Distrib. Comput. **20**(1), 3–19 (2007)
2. Datta, A.K., Larmore, L.L., Masuzawa, T., Sudo, Y.: A self-stabilizing minimal k-grouping algorithm. In: Proceedings of the 18th International Conference on Distributed Computing and Networking, pp. 3:1–3:10. ACM (2017)
3. Datta, A.K., Larmore, L.L., Masuzawa, T.: Constant space self-stabilizing center finding in anonymous tree networks. In: Proceedings of the International Conference on Distributed Computing and Networking, pp. 38:1–38:10 (2015)
4. Peleg, D., Upfal, E.: The token distribution problem. SIAM J. Comput. **18**(2), 229–243 (1989)