A Communication-Efficient Self-stabilizing Algorithm for Breadth-First Search Trees

Ajoy K. Datta[1], Lawrence L. Larmore[1], and Toshimitsu Masuzawa[2]

[1] Department of Computer Science, University of Nevada, Las Vegas, USA
{ajoy.datta,lawrence.larmore}@unlv.edu
[2] Graduate School of Information Science and Technology, Osaka University, Japan
masuzawa@ist.osaka-u.ac.jp

Abstract. A self-stabilizing algorithm converges to its designated behavior from an arbitrary initial configuration. It is standard to assume that each process maintains communication with all its neighbors. We consider the problem of self-stabilizing construction of a breadth first search (BFS) tree in a connected network of processes, and consider algorithms which are not given the size of the network, nor even an upper bound on that size. It is known that an algorithm that constructs a BFS tree must allow communication across every edge, but not necessarily in both directions. If m is the number of undirected edges, and hence the number of directed edges is $2m$, then every self-stabilizing BFS tree algorithm must allow perpetual communication across at least m directed edges. We present an algorithm with reduced communication for the BFS tree problem in a network with unique identifiers and a designated root. In this algorithm, communication across all channels is permitted during a finite prefix of a computation, but there is a reduced set of directed edges across which communication is allowed forever. After a finite prefix, the algorithm uses only $m+n-1$ directed edges for communication, where n is the number of processes in the network and m is the number of edges.

1 Introduction

A *self-stabilizing distributed system* [8] can eventually recover its intended behavior without external intervention even when started from an arbitrary configuration (or global state). Thus, a self-stabilizing distributed system attains high tolerance to transient faults and high adaptability to dynamic topology changes of networks. Self-stabilization is usually implemented by combining a *transient-fault detection* mechanism and a *fault correction* mechanism. The former guarantees that an alarm is raised at a process if the distributed system is at an illegitimate configuration. The latter is initiated by the alarmed process to bring the system to a legitimate configuration.

The fault correction mechanism can cost dearly in time and/or communication, since it must make the distributed system recover from *any possible* configuration.

Thus, the prime concern in efficiency of self-stabilization considered so far is efficiency of the fault correction mechanism, for example, the maximum time (*stabilization time*) required to recover legitimacy from any configuration, and the maximum number of bits exchanged until the system reaches a legitimate configuration from any configuration.

The cost of the transient-fault detection mechanism is another crucial difference in cost between self-stabilizing protocols and non-self-stabilizing classical protocols, since the latter (starting from predetermined initial configurations) need no transient-fault detection. In a self-stabilizing protocol, the transient-fault detection mechanism requires each process to keep communicating forever with some of its neighboring processes to check consistency; otherwise, a process may initially start and remain forever at a state inconsistent with those of its neighboring processes, and the protocol cannot recover legitimacy.

Cost of the transient-fault detection mechanism has not caused much concern so far. Most self-stabilizing protocols proposed up to now require every process to communicate with all of its neighboring processes repeatedly and forever to check consistency among them. This leads to a high communication load in networks and makes self-stabilizing protocols unacceptable in some real situations.

It is worth noting that efficiency of transient-fault detection mechanism is one of the most important concerns of self-stabilization. In self-stabilizing distributed systems, the transient-fault detection mechanism operates almost all the time, but the fault correction mechanism operates only when necessary. Thus, efficiency of the transient-fault detection mechanism normally dominates the efficiency of self-stabilizing distributed systems.

Our Contribution. In this paper, we present a silent and self-stabilizing communication-efficient algorithm, ROOT-UID, for constructing a *breadth-first search* (BFS) tree in a connected network G with the UID property, *i.e.,* where processes have unique IDs, and where there is a designated root process, *Root*. Throughout this work, we let n be the number of processes in our network, and m the number of edges. ROOT-UID is $\Diamond$-$(m + n - 1)$-communication-efficient, meaning that, eventually, communication is needed across only $m + n - 1$ of the $2m$ directed edges of the network. More specifically, there is 2-way communication forever across each of the BFS tree edges, but communication in only one direction across each cross edge. ROOT-UID uses the identifiers to decide which direction is used: if $\{x, y\}$ is a cross edge (*i.e.,* an edge which does not connect a process with its parent) and $x.id > y.id$ then, eventually, x can read the variables of y, but y cannot read the variables of x.

We introduce two techniques designed to cope with the peculiar problems of communication efficient computation. We introduce the *three color control scheme*, which is used to "wake up" processes when necessary, and "put them back to sleep" when their job is done. This scheme is important because during some finite prefix of a computation, processes may need to read all their neighbors, but after that prefix, only a subset of neighbors.

We also introduce the concept of "net polarity" by which ROOT-UID detects whether the putative BFS tree contains all processes of the network. Suppose, for example, that, at the initial configuration (which is arbitrary) there is a *false BFS tree*, a subgraph T' rooted at *Root*, which locally looks exactly like a BFS tree, but does not contain all processes. Yet because it "looks good" locally, all processes of T' "think" that a final configuration has been reached, and are resting. In order to restart the BFS tree construction phase of the algorithm, some process of T' must become aware that T' does not contain all processes. We further suppose that the ID of every process in T' is smaller than the ID of any of its neighbors not in T', which implies that, since all processes of T' are resting, none of them can read any neighbor not in T'. The problem is, how can the resulting deadlock be broken? That is, how can the processes of T' detect that T' does not contain all processes in the network?

Related Work. Anguilera *et al.* [1] introduce the concept of *communication efficiency* in implementation of a failure detector Ω[1] in partially synchronous systems. Following their work, there was further investigation of the possibility of communication efficient implementations of failure detector Ω (*e.g.,* [2,3,4,11]). The aim of communication-efficiency is to reduce the number of indefinitely communicating process pairs [1,2,3,11], and to reduce the number of processes that broadcast indefinitely [4].

Dolev and Schiller [9] introduce the *communication adaptive* property of self-stabilizing protocols. A self-stabilizing protocol is communication adaptive if the communication load of the transient-fault detection mechanism is low, while that of the fault correction mechanism is high. They present a communication adaptive self-stabilizing protocol for group membership service. Its communication complexity per asynchronous cycle is $O(nm \log N)$ bits before convergence to a legitimate configuration, and reduces to $O(n^2 \log N)$ bits after convergence, where n and m are the numbers of processes and links respectively, and N is an upper bound on the number of processes.

Delporte-Gallet *et al.* [6] consider self-stabilizing leader election that can tolerate process crashes as well as transient faults. They present an algorithm in the fully-synchronous system that uses only $n-1$ unidirectional links to carry messages repeatedly and forever.

Devismes *et al.*[7] introduce communication efficiency with a *local* criterion. They consider, as communication-efficiency, the number of neighbors that each process communicates with forever as well as the total number of communicating process pairs. They investigate communication-efficiency for the maximal independent set problem and the maximal matching problem.

The most closely related previous work is [13], which considers communication-efficiency of self-stabilizing spanning-tree construction and gives possibility and impossibility results. They show that there exists a self-stabilizing communication-efficient algorithm for spanning tree construction, where only $n-1$ process pairs maintain communication indefinitely, provided a unique root is designated. Their algorithm constructs an arbitrary spanning tree, not a BFS tree.

[1] Roughly speaking, the failure detector Ω eventually provides all processes with the identifier of a unique correct process (*i.e.,* a leader).

In the same paper, Masuzawa *et al.*give an important lower bound. If there is no designated root process, even if the UID property holds, any communication efficient self-stabilizing algorithm for spanning tree construction must use every edge in one or the other direction, yielding a lower bound of m on the communication efficiency of spanning tree construction when there is no designated root, where m is the number of edges in the network.

Takimoto *et al.* [14] consider communication-efficiency in wireless networks where a communication primitive is a local broadcast that allows a process to send a message to all of its neighboring processes. They aim to reduce the number of processes that keep broadcasting forever. The results of [13] and [14] are also summarized in [12].

Kutten and Zinenko [10] investigate the possibility of self-stabilizing protocols that are communication-efficient, both during and after convergence to legitimacy. They use *randomness* to achieve communication efficiency, and present communication efficient self-stabilizing protocols for spanning tree construction, distributed reset, and unison.

Outline. In Section 2, we define our model, and describe some of the common features of our algorithms. In Section 3, we present our algorithm ROOT-UID, a communication-efficient algorithm for constructing a BFS tree in a connected network with the UID property, given that the network has a distinguished root process. In 3.4 we explain the *three color control scheme*, which is used during error recovery. In Section 3.6, we explain the need for *net polarity* variables, which we use to ensure that each edge is used for communication in at least one direction. Section 4 concludes the paper.

2 Preliminaries

2.1 Self-stabilization

We use the composite atomicity shared memory model of computation [8]. The *program* of each process consists of a finite set of *actions* of the following form: $<$ *label* $>$ $<$ *guard* $>$ $\longrightarrow$ $<$ *statement* $>$. The *guard* of an action in the program of a process x is a Boolean expression involving the variables of x and its neighbors. The *statement* of an action of x updates one or more variables of x. An action can be executed only if it is *enabled*, *i.e.,* its guard evaluates to true. A process is said to be *enabled* if at least one of its actions is enabled. We use the *distributed daemon.* If one or more processes are enabled, the daemon *selects* at least one of these enabled processes to execute an action. We also assume that daemon is *unfair*, *i.e.,* it selects an arbitrary non-empty set of enabled processes at a step, if there is at least one. Thus, the daemon need never select a given enabled process unless it becomes the only enabled process. We measure time complexity in *rounds* elapsed before the first legitimate configuration. A *round* is defined to be a minimal computation sequence during which every process initially enabled is selected or becomes disabled by the end of the round [8].

Self-Stabilization. We say that an algorithm $\mathcal{A}$ for a problem $\mathbb{P}$ is *self-stabilizing* if there is a given class of configurations of $\mathcal{A}$, which we call the *legitimate* configurations of $\mathcal{A}$, such that the following conditions hold: (i) **Closure:** If a computation of $\mathcal{A}$ starts in a legitimate configuration, all subsequent configurations of that computation are legitimate. (ii) **Convergence:** Starting from an arbitrary configuration, the configuration of any computation of $\mathcal{A}$ contains a legitimate configuration. (iii) **Correctness:** If configuration is legitimate, the output conditions of $\mathbb{P}$ are satisfied. Note that Closure and Convergence together imply that every computation is eventually legitimate. We say that a configuration of a distributed algorithm $\mathcal{A}$ is *final* if, at that configuration, no process is enabled to execute any action of $\mathcal{A}$. We say that a self-stabilizing distributed algorithm $\mathcal{A}$ is *silent* if every computation of $\mathcal{A}$ contains a legitimate final configuration.

2.2 Communication Efficiency

Informally, an algorithm is *communication efficient* [7,13] if, eventually, not all links are used for communication between processes. We say that a computation of $\mathcal{A}$ is r-communication-efficient if the number of *directed* edges of the network used by the computation does not exceed r. That is, for every process x there is a set $R(x) \subseteq N(x)$, where $N(x)$ is the set of neighbors of x, such that x reads only its own variables and those of members of $R(x)$ during the computation, where $\sum |R(x)| \leq r$. We say that an algorithm $\mathcal{A}$ is $\diamondsuit$-r-communication-efficient, or *eventually* r-communication-efficient, if every computation of $\mathcal{A}$ has a suffix which is r-communication-efficient.

In an implementation of a distributed algorithm, not only must a process which executes an action have evaluated the guard of that action to TRUE, but processes which do *not* execute must evaluate guards of all actions to FALSE. More specifically, at any particular step, if correctness of $\mathcal{A}$ depends on a process x not executing a particular action, x must evaluate the guard of that action to be FALSE. If the value of that guard cannot be computed without using the values of a neighboring process y, then x must read y at that step. This fact imposes an interesting condition on a communication efficient algorithm. In any computation, except for a finite prefix, every process must evaluate every guard using only its own variables and those of $R(x)$, rather than the variables of all neighbors.

3 BFS Tree Computation with Distinguished Root and UID

In this section, we give a distributed algorithm, ROOT-UID, which constructs a BFS tree in a connected network G with the UID property and a distinguished process. ROOT-UID is silent and self-stabilizing under the unfair daemon, converges in $O(n)$ rounds, and is $\diamondsuit$-$(m+n-1)$-communication efficient. Eventually, each tree edge is used in both directions, but each cross edge is used in just one direction. More specifically, if $\{x, y\}$ is a cross edge, and $x.id < y.id$, then $x \in R(y)$ and $y \notin R(x)$, *i.e.,* eventually, y reads x, but x does not read y.

3.1 Simple BFS Construction

ROOT-UID is a communication efficient implementations of the algorithm SIMPLE-TREE given below, a silent self-stabilizing BFS tree construction algorithm for any connected network with a distinguished root, which has only two variables, *level* and *parent*. We assume that $Root.levl = 0$ and $Root.parent = \perp$ are fixed. SIMPLE-TREE is not communication efficient; each process reads all of its neighbors at every step.

Figure 3.1: Code of SIMPLE-TREE **for process** $x \neq Root$.

```
Function: $Level(x) = 1 + \min\{y.level : y \in N(x)\}$
Actions:
3.1.1.  $x.level \neq Level(x)$  $\longrightarrow x.level \leftarrow Level(x)$
3.1.2.  $(x.level \neq 1 + x.parent.level) \wedge (y \in N(x)) \wedge (1 + y.level = x.level)$  $\longrightarrow x.parent \leftarrow y$
```

Remark 1. On a connected network G with a distinguished process *Root*, SIMPLE-TREE is silent and self-stabilizing, and converges in $O(d)$ rounds, where d is the diameter of G.

Proof Sketch. After t rounds, $x.level = ||x, Root||$ if $||x, Root|| \leq t$, and within one more round, $x.parent$ will be correct. □

3.2 Variables and Functions of ROOT-UID

In ROOT-UID, each process x has the following variables.

1. $x.parent \in N(x) \cup \{\perp\}$.
2. $x.level$, non-negative integer, the *level* of x, which is eventually equal to the distance from x to the root of the BFS tree.
3. $x.color \in \{0, 1, 2\}$, the *color* of x, which is used in the three-color control scheme, which we describe in detail in Section 3.4. If $x.color = 0$, then x is *resting*, while x is *alert* if $x.color \in \{1, 2\}$.
4. $x.polarity[y] \in \{-1, 1\}$, the *polarity* of the directed edge (x, y), for each $y \in N(x)$. We say that y is a *restricted* neighbor of x is $x.polarity[y] = -1$, otherwise y is an *unrestricted* neighbor.
5. $x.is\_child[y]$ for $y \in N(x)$, a Boolean array, where $x.is\_child[y]$ means that y is a child of x.
   Both $x.polarity[\,]$ and $x.is\_child[\,]$ require $O(\delta_x)$ space, where δ_x is the degree of x.
6. $x.loc\_net\_polarity$, integer, which can be positive, negative, or zero. Eventually, the value of this variable is $\sum_{y \in N(x)} x.polarity[y]$.
7. $x.net\_polarity$, integer. Eventually, $x.net\_polarity = \sum_{y \in T_x} y.loc\_net\_polarity$ for all x, where T_x is the subtree of T rooted at x, and $Root.net\_polarity = 0$,

We now list the functions of ROOT-UID, each of which (except *Root*) is defined for one process x, or for two processes x and $y \in N(x)$. Some functions have names which are capitalized versions of variable names. In each of those cases, the function returns the corrected value of the corresponding variable.

8. *Root*, a designated process. Let $Root.parent = \perp$ be fixed, and $Root.level = 0$ be fixed.
9. $Is\_Child(x, y) \equiv (y.parent = x) \wedge (y.level = 1 + x.level)$, Boolean, meaning that y is a child of x.
10. $Chldrn(x) = \{y \in N(x) : x.is\_child[y]\}$, the set of children of x.
11. $Family(x) = \{y \in N(x) : x.is\_child[y] \vee x.parent = y\}$
12. $False\_Root(x) \equiv (x \neq Root) \wedge ((x.parent = \perp) \vee (x.level \neq 1 + x.parent.level))$, Boolean.
13. $Is\_Root(x) \equiv (x = Root) \vee False\_Root(x)$, Boolean, namely x is a *root* of the forest T. At any step in the algorithm, T is the directed graph whose edges are all (x, y) such that $Is\_Child(y, x)$. Thus T has out-degree at most 1 at each process, and cannot contain a cycle, since the variable $x.level$ is decreasing along any directed path, and thus T is a forest.
14. $Level(x) = \begin{cases} 0 \text{ if } Is\_Root(x) \\ 1 + \min\{y.level : y \in N(x)\} \text{ otherwise} \end{cases}$
15. $Polarity(x, y) = \begin{cases} 1 \text{ if } y.id < x.id \\ -1 \text{ if } y.id > x.id \end{cases}$
16. $Loc\_Net\_Polarity(x) = \sum_{y \in N(x)} x.polarity[y]$, integer.
17. $Net\_Polarity(x) = x.loc\_net\_polarity + \sum_{y \in Chldrn(x)} y.net\_polarity$, integer.
18. $Nbr\_Ok(x, y) = \Big(x.polarity[y] = Polarity(x, y)\Big) \wedge \Big(x.is\_child[y] = Is\_Child(x, y)\Big) \wedge \Big(|x.level - y.level| \leq 1\Big)$, Boolean, meaning that y appears, to x, to have values consistent with legitimacy.
19. $Unrestricted\_Nbrs(x) = Family(x) \cup \{y \in N(x) : x.polarity[y] = 1\}$, the set of neighbors that can be read by x when x is resting.
20. $Visible\_Nbrs(x) = \begin{cases} Unrestricted\_Nbrs(x) \text{ if } x.color = 0 \\ N(x) \text{ otherwise} \end{cases}$
    the neighbors of x that x can read, given its current state.
21. $Visible\_Nbrs\_Ok(x) \equiv (\forall y \in Visible\_Nbrs(x))\ Nbr\_Ok(x, y)$, Boolean.
22. $Ok(x) \equiv Visible\_Nbrs\_Ok(x) \wedge \neg False\_Root(x) \wedge (x = Root \Rightarrow x.net\_polarity = 0)$ Boolean.
    The clause that $Root.net\_polarity = 0$ is explained in Section 3.6.
23. $Color(x) = \begin{cases} 0 \text{ if } & Ok(x) \wedge \Big((x = Root) \vee (x.parent.color = 0)\Big) \wedge \\ & \forall y \in Chldrn(x)(y.color \neq 1) \\ 1 \text{ if } \neg Ok(x) \vee \exists y \in Chldrn(x)(y.color = 1) & \\ 2 \text{ if } & Ok(x) \wedge (x.parent.color \neq 0) \wedge \forall y \in Chldrn(x)(y.color \neq 1) \end{cases}$

Additional notation which is common to the rest of this paper includes

- T_x, the subtree of T rooted at x.
- $\delta_x = |N(x)|$, the *degree* of x.
- Tree edge, an undirected edge of G that connects some process x with $x.parent$.
- Cross edge, an undirected edge of G that is not a tree edge.
- $Cross(x) = N(x) \setminus Family(x)$, the *cross neighbors* of x.

3.3 Actions of ROOT-UID

We classify the actions of ROOT-UID as *easy* or *hard*. The guard of an easy action can be evaluated only by examining unrestricted neighbors, while evaluation of the guard of a hard action may require examining all neighbors. If a process x is resting, meaning that $x.color = 0$, then x cannot be enabled to execute a hard action, since one of the clauses of the guard of every hard action is that $x.color \neq 0$, *i.e.,* x is alert.

Figure 3.2: Code of ROOT-UID **for one process x and $y \in N(x)$.**

```
Easy Actions:
3.2.1. $x.color \neq Color(x)$                              $\longrightarrow x.color \leftarrow Color(x)$
3.2.2. $x.loc\_net\_polarity \neq Loc\_Net\_Polarity(x)$    $\longrightarrow x.loc\_net\_polarity$
                                                                $\leftarrow Loc\_Net\_Polarity(x)$
3.2.3. $x.net\_polarity \neq Net\_Polarity(x)$              $\longrightarrow x.net\_polarity \leftarrow Net\_Polarity(x)$
Hard Actions:
3.2.4. $x.color \neq 0 \ \wedge \ x.level \neq Level(x)$    $\longrightarrow x.level \leftarrow Level(x)$
3.2.5. $x.color \neq 0 \ \wedge \ x.level \neq 1 + x.parent.level \ \wedge$   $\longrightarrow x.parent \leftarrow y$
       $y \in N(x) \ \wedge \ 1 + y.level = x.level$
3.2.6. $x.color \neq 0 \ \wedge \ y \in N(x) \ \wedge$      $\longrightarrow x.polarity[y] \leftarrow Polarity(x,y)$
       $x.polarity[y] \neq Polarity(x,y)$
3.2.7. $x.color \neq 0 \ \wedge \ y \in N(x) \ \wedge$      $\longrightarrow x.is\_child[y] \leftarrow Is\_Child(x,y)$
       $x.is\_child[y] \neq Is\_Child(x,y)$
```

Figure 3.2 gives the actions of ROOT-UID. We define a configuration of ROOT-UID to be *legitimate* if T is a BFS tree, and if the variables $x.polarity[y]$, $x.is\_child[y]$, $x.loc\_net\_polarity$, and $x.net\_polarity$ have the correct values for all x and y, and if $x.color = 0$ for all x.

Construction of the BFS tree T is done by Actions 3.2.4 and 3.2.5, both of which are hard actions. The purpose of the three color control structure is to ensure that there is enough communication to enable that construction, and also to ensure that, once that construction is finished, the algorithm is $(m+n-1)$-communication efficient. Action 3.2.6 is executed at most once for each ordered pair (x,y), since $Polarity(x,y)$ never changes. $\sum_{x\in G}\sum_{y\in N(x)} Polarity(x,y) = 0$, since $Polarity(x,y) + Polarity(y,x) = 0$ for every edge $\{x,y\}$. Actions 3.2.2 and 3.2.3 are both easy, and are bottom-up waves which cause $Root.net\_polarity$ to be set to $\sum_{x\in G}\sum_{y\in N(x)} x.polarity[y]$ which will be zero if all values of $x.polarity[y]$ are correct. If not, $Ok(Root) =$ FALSE, and the three color control scheme causes all processes to become alert, ensuring that polarities become correct.

3.4 The Three Color Scheme

We now describe in detail how the three color scheme is used in ROOT-UID. We define the predicate *Enabled*_Hard(x) to mean that either x is enabled to execute a hard action, or that x is resting and would be enabled to execute a hard action if $x.color$ were changed. In order for the three color structure scheme to work, Property 1, given below, must hold.

Property 1. If the current configuration is illegitimate and legitimacy cannot be achieved by easy actions alone, then there exist processes x, y such that $\neg Ok(x) \wedge \mathit{Enabled}$_Hard$(y)$ either holds or will hold at some later step, and x, y lie in the same component of T.

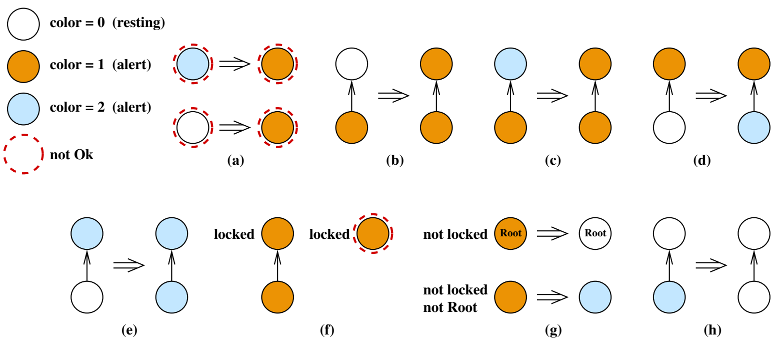

Figure 3.3: Some examples of color actions. In each case, the value of $x.color$ changes to match $Color(x)$.

We say that a process x is *locked* if $x.color = 1$, and if either $\neg Ok(x)$ or $y.color = 1$ for some $y \in Chldrn(x)$. Note that a locked process is not enabled to change its color. Suppose all processes have color 0, and $Ok(x) =$ FALSE for some x. Then x executes Action 3.2.1; $x.color \leftarrow 1$, and x becomes locked. In a convergecast wave, every ancestor of x, including *Root*, changes color to 1 and becomes locked. A broadcast wave, initiated by *Root* and by every other process of color 1, changes all processes to color 2 except for those which already have color 1 or 2.

The entire path between x and *Root* will remain in color 1, and locked, and all processes will remain alert as long as $\neg Ok(x)$ holds. When *Ok* holds for all processes, all color 1 processes except *Root* will change color to 2 in a bottom-up wave, after which *Root* will change color to 0. In a broadcast wave, all processes will then change color to 0. After possibly more executions of easy actions, legitimacy will be achieved.

Figure 3.3 shows the effect of Action 3.2.1 in various situations. 3.3(a) shows a convergecast color 1 wave initiated by a process x whose current color is either 0 or 2, where $Ok(x) =$ FALSE. Then $x.color \leftarrow 1$, starting the wave. The wave is propagated upward, as shown in 3.3(b) and 3.3(c). Figure 3.3(d) shows how a process in color 1 initiates a broadcast wave which alerts all processes below it. In this figure, a resting (color 0) process has a parent of color 1, and becomes alert by changing color to 2. Propagation of that wave downward is illustrated in 3.3(e).

The next three parts of Figure 3.3 deal with restoring the resting state. Figure 3.3(f) illustrates locked processes. Figure 3.3(g) shows propagation of the convergecast wave which eliminates color 1. If a color 1 process is unlocked, indicating that its original purpose has been fulfilled, it changes color to 2, unless it is *Root*, in which case it changes color to 0. Once $Root.color = 0$, a broadcast wave will change the color of all processes to 0. (Unless, of course, there is a new instance of a process where *Ok* does not hold.) The propagation of that wave is illustrated in Figure 3.3(h).

3.5 Example Computation Showing the Three Color Scheme

In Figure 3.4, each ID is the letter shown inside the circle representing the process. (The ID of *Root* is not relevant.) Parent pointers are shown as arrows, and cross edges as dashed lines. The figure shows an example computation of ROOT-UID on a network where $n = 7$ and $m = 7$.

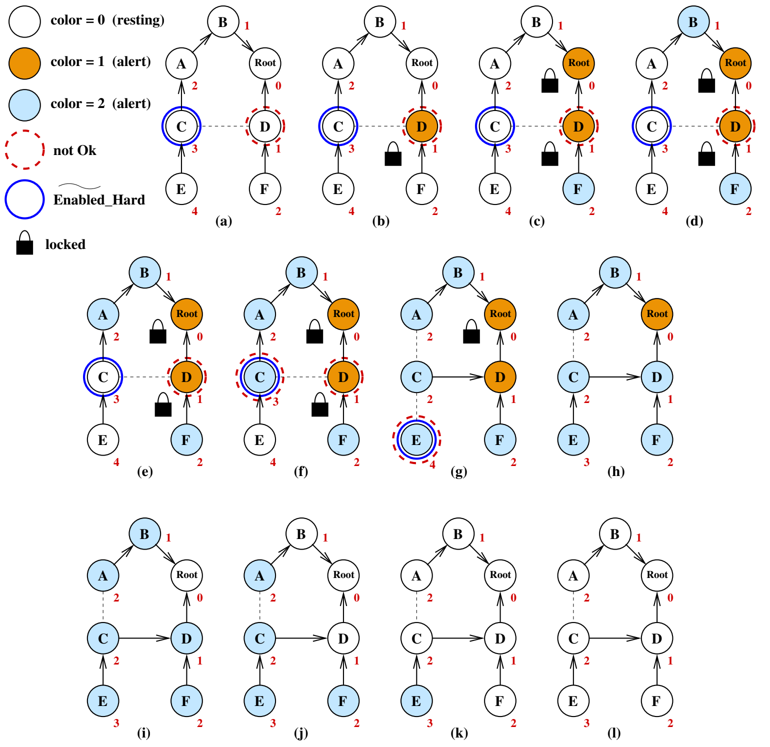

Figure 3.4: Example Computation of ROOT-UID. The links of the spanning tree T are shown as arrows, while other edges of G are shown as dashed lines. Values of *level* are shown in red. Initially, T is not a BFS tree. Figure 3.4 (l) shows a configuration in which T is correct and all processes are resting.

In the initial configuration, shown in Figure 3.4(a), all processes are resting. T is not correct, since C.*parent* should be D. The solid outer circle around C indicates that $\widetilde{Enabled}$_Hard(C) = TRUE, while the dashed circle indicates that

Ok(D) = FALSE. It may seem strange that action by C is required to "fix" the configuration, but that D, which is far from C in the tree, is the only process aware of this need. This situation is not unusual. In this case, since C < D, D can see that C.*level* = 3, which is inconsistent with its own level, which is 1. On the other hand, C does not see D, and hence does not notice the inconsistency.

In 3.4(b), D initiates a color 1 convergecast wave, which reaches *Root* in Figure 3.4(c). The resulting color 2 broadcast wave, which is initiated by both D and *Root*, eventually reaches all processes. As soon as the wave reaches C, in 3.4(f), that process executes a hard action, changing C.*parent* to D and C.*level* to 2. The configuration is still not legitimate, since now E.*level* = 4. In this case, there is a circle of each color around E, since *Ok*(E) = FALSE, and E is also enabled to execute a hard action, which it does in the next step, changing its level to 3. Meanwhile, since D is no longer locked in Figure 3.4(g), it changes its color to 2. If there were more color 1 processes between D and *Root*, their color would change from 1 to 2 in a bottom-up wave. *Root* changes color from 1 directly to 0 at 3.4(i), initiating a broadcast wave that causes all processes to change color to 0, as shown in Figure 3.4(l).

3.6 The Purpose of Polarity

The purpose of the variables $\left\{x.\mathit{polarity}[y]\right\}$ is to tell x whether y is a restricted neighbor, information that x must first obtain by reading y. But that presents a problem: what if both x and y are initialized in such a way that each believes the other to be a restricted neighbor? In that case, one of them has the wrong information, but neither knows it.

We solve that problem by using the *net polarity* of T, $\sum_{x \in T} \sum_{y \in N(x)} x.\mathit{polarity}[y]$. If all values of the polarity variables are correct, this net polarity will be zero, since each edge contributes 1 at one end and -1 at the other. The net polarity is computed bottom up, and the total value is stored at *Root.net_polarity*. If that value is not zero, there is an error, and *Root* will send a broadcast wave, alerting all processes. After sufficiently many actions, both easy and hard, T will be correct.

Figure 3.5 illustrates an example which shows the necessity of the net polarity variables. Suppose the variables *loc_net_polarity* and *net_polarity* as well as Actions 3.2.2 and 3.2.3 are removed from the definition of ROOT-UID, but all other variables and actions remain. In this case, there is a configuration, illustrated in Figure 3.5(a) which is both illegitimate and final, *i.e.,* a deadlock.

In Figure 3.5, each ID is the letter shown inside the circle representing the process, and E = *Root*. Parent pointers are shown as arrows, and cross edges as dashed lines. The "+" and "−" signs at the ends of each edge indicate polarity; if $y \in N(x)$, a plus sign indicates that $x.\mathit{polarity}[y] = 1$, meaning that x can read y when x is resting, and a minus sign indicates that $x.\mathit{polarity}[y] = -1$, meaning that x must be alert in order to read y. Note that $x.\mathit{polarity}[y] = 1 \Leftrightarrow x.\mathit{id} > y.\mathit{id}$ in a legitimate configuration. In 3.5(a), there are two places where the legitimacy fails: D.*polarity*[B] = -1 when it should be 1, and B.*parent* = G when it should be D. Both processes are enclosed with solid circles, indicating that they would

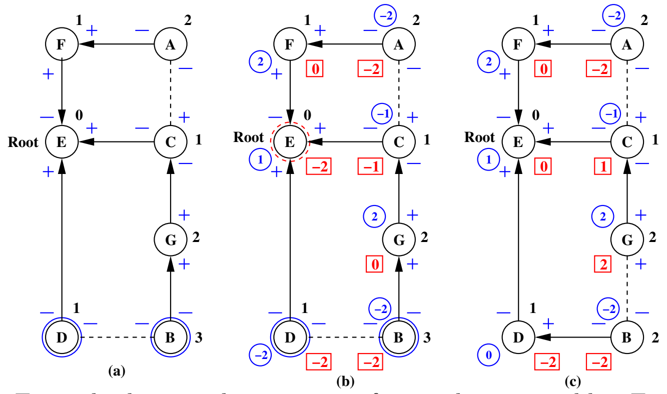

Figure 3.5: Example showing the necessity of net polarity variables. E = *Root*. Tree edges are black arrows, cross edges are dashed, and levels are shown as numerals. A "+" or "−" near x on the edge between x and y indicates that $x.\mathit{polarity}[y]$ is 1 or −1, respectively.

be enabled to execute a hard action if they became alert. B would be enabled to execute Action 3.2.5 if it were alert, while D would be enabled to execute Action 3.2.6 if it were alert. Neither action is enabled, since both processes are resting , *i.e.,*B.*color* = D.*color* = 0; in fact no process is enabled. The configuration is a deadlock.

In Figure 3.5(b), we illustrate the same configuration, but with the values of *loc_net_polarity* and *net_polarity* filled in. The value of x.*loc_net_polarity* is shown as a numeral enclosed in a solid circle, while the value of x.*net_polarity* is shown as a numeral enclosed in a box. Note that E.*net_polarity* = −2, which means that *Ok*(E) = FALSE. since *Net_Polarity*(*Root*) must be zero in a legitimate configuration. (Conversely, if *Net_Polarity*(*Root*) were zero, it would indicate legitimacy, since if both end processes of an edge have positive polarity, that error would be detected.) The remaining steps of the computation are not shown, other than the final configuration at the end. E changes color to 1, triggering a broadcast wave in which all other processes change color to 2, which means they are alert and able to execute the needed actions, after which the configuration is legitimate, as shown in Figure 3.5(c). We list the sequence of changes below.

(a)	B.*parent* ← *D*	(b)	B.*level* ← 2	(c)	D.*polarity*[B] ← 1
(d)	D.*loc_net_polarity* ← 0	(e)	D.*net_polarity* ← −2	(f)	G.*net_polarity* ← 2
(g)	C.*net_polarity* ← 1	(h)	E.*net_polarity* ← 0		

3.7 Proof Sketches for ROOT-UID

Lemma 1. *The number of steps of any computation of* ROOT-UID *at which any hard action is executed is finite.*

Proof Sketch. A process can execute Action 3.2.6 at most once in a computation. We prove by induction on $||x, \mathit{Root}||$ that a process x can only execute Action

3.2.4 finitely many times; as a corollary, we have that x can only execute Action 3.2.5 finitely many times. □

Lemma 2. *There is no infinite computation of* ROOT-UID *during which no hard action is executed.*

Proof Sketch. Suppose Γ is an infinite computation of ROOT-UID, during which no process executes a hard action. The shape of T does not change, and since Action 3.2.6 is not executed, the value of $x.polarity[y]$ for any x and y does not change. Therefore, no process can execute Action 3.2.2 more than once, and hence there is a last step at which any process executes 3.2.2. After the last action of Action 3.2.2, the values of $x.loc\_net\_polarity$ do not change, and thus no process can execute Action 3.2.3 more than once thereafter, hence there is a last step at which any process executes 3.2.3. After the last action of 3.2.3, the value of $Ok(x)$ does not change for any x. Thus, there is an infinite computation during which Action 3.2.1 is executed infinitely many times. Finally, we obtain a contradiction by defining a non-negative integral potential which decreases at any step at which Action 3.2.1 is executed, but no other action is executed. Since the potential cannot be less than zero, the computation is finite. □

Lemma 3. *Any configuration of* ROOT-UID *that is final is also legitimate.*

Proof Sketch. Let γ be an illegitimate configuration of ROOT-UID. We break into cases, and show that in each case, there is some process that is enabled to execute an action, and thus γ is not final. □

Theorem 1. ROOT-UID *is $\diamondsuit$-$(m+n-1)$-communication efficient and is silent and self-stabilizing under the unfair daemon, reaches a legitimate configuration within $O(n)$ rounds, and uses $O(\log n + \delta_x)$ space per process.*

Proof Sketch. The communication efficiency of ROOT-UID follows from Lemmas 1, 2, and 3.

In the initial configuration, the longest chain in T is no longer than $n-1$. If the configuration is not final, every process will be alerted within $O(n)$ rounds. Thereafter, convergence will occur within $O(d)$ rounds in the same manner as in SIMPLE-TREE. Thus, ROOT-UID reaches a legitimate configuration within $O(n)$ rounds.

The value of $x.level$ takes $O(\log d)$ space, and x must retain $O(1)$ memory for each neighbor y to hold the value of $x.polarity[y]$. The value of $x.net\_polarity$ is an integer whose absolute value cannot exceed m, and no other variable requires more space. Thus, the per process space complexity of ROOT-UID is $O(\log n + \delta_x)$. □

4 Conclusion

We have given a self-stabilizing and silent algorithm, ROOT-UID, for BFS construction in a network with the UID property which has a designated root process, which is $\diamondsuit$-$(m + n - 1)$-communication efficient, and which takes $O(n)$ rounds to reach legitimacy. The space complexity of ROOT-UID is $O(\log n + \delta_x)$ for each process x.

Acknowledgement. This work was supported in part by JSPS KAKENHI, Grant Numbers (B)26280022 and 24650012.

References

1. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Stable leader election. In: Welch, J. (ed.) DISC 2001. LNCS, vol. 2180, pp. 108–122. Springer, Heidelberg (2001)
2. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: On implementing omega with weak reliability and synchrony assumptions. In: Proceedings of the 22rd ACM Symposium on Principles of Distributed Computing, pp. 306–314 (2003)
3. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Communication-efficient leader election and consensus with limited link synchrony. In: Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing, pp. 328–337 (2004)
4. Biely, M., Widder, J.: Optimal message-driven implementations of omega with mute processes. ACM Transactions on Autonomous and Adaptive Systems 4(1), 4:1–4:22 (2009)
5. Datta, A.K., Larmore, L.L., Vemula, P.: Self-stabilizing leader election in optimal space under an arbitrary scheduler. Theoretical Computer Science 412(40), 5541–5561 (2011)
6. Delporte-Gallet, C., Devismes, S., Fauconnier, H.: Robust stabilizing leader election. In: Masuzawa, T., Tixeuil, S. (eds.) SSS 2007. LNCS, vol. 4838, pp. 219–233. Springer, Heidelberg (2007)
7. Devismes, S., Masuzawa, T., Tixeuil, S.: Communication efficiency in self-stabilizing silent protocols. In: Proceedings of the 29th International Conference on Distributed Computing Systems, pp. 474–481 (2009)
8. Dolev, S.: Self-stabilization. MIT Press (2000)
9. Dolev, S., Schiller, E.: Communication adaptive self-stabilizing group membership service. In: IEEE Transactions on Parallel and Distributed Systems, pp. 709–720 (2003)
10. Kutten, S., Zinenko, D.: Low communication self-stabilization through randomization. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 465–479. Springer, Heidelberg (2010)
11. Larrea, M., Fernandez, A., Arevalo, S.: Optimal implementation of the weakest failure detector for solving consensus. In: Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems, pp. 52–59 (2000)
12. Masuzawa, T.: Silence is golden: Self-stabilizing protocols communication-efficient after convergence. In: Défago, X., Petit, F., Villain, V. (eds.) SSS 2011. LNCS, vol. 6976, pp. 1–3. Springer, Heidelberg (2011)
13. Masuzawa, T., Izumi, T., Katayama, Y., Wada, K.: Brief Announcement: communication-efficient self-stabilizing Protocols for spanning-tree construction. In: Abdelzaher, T., Raynal, M., Santoro, N. (eds.) OPODIS 2009. LNCS, vol. 5923, pp. 219–224. Springer, Heidelberg (2009)
14. Takimoto, T., Ooshita, F., Kakugawa, H., Masuzawa, T.: Communication-efficient self-stabilization in wireless networks. In: Richa, A.W., Scheideler, C. (eds.) SSS 2012. LNCS, vol. 7596, pp. 1–15. Springer, Heidelberg (2012)