Self-Stabilizing Labeling and Ranking in Ordered Trees[☆]

Ajoy K. Datta[a], Stéphane Devismes[b], Lawrence L. Larmore[a], Yvan Rivierre[b]

[a] *School of Computer Science, University of Nevada Las Vegas*
[b] *VERIMAG UMR 5104, Université Joseph Fourier, Grenoble*

Abstract

We give two self-stabilizing algorithms for tree networks. The first computes an index, called *guide pair*, for each process P in $O(h)$ rounds using $O(\delta_P \log n)$ space per process, where h is the height of the tree, δ_P the degree of P, and n the number of processes in the network. Guide pairs have numerous applications, including ordered traversal or navigation in the tree. Our second algorithm, which uses the guide pairs computed by the first algorithm, solves in $O(n)$ rounds the *ranking problem* for an ordered tree, where each process has an *input value*. This second algorithm has space complexity $O(b+\delta_P \log n)$ in each process P, where b is the number of bits needed to store an input value. The first algorithm orders the tree processes according to their topological positions. The second algorithm orders (ranks) the processes according to their input values.

Keywords: Self-stabilization, tree networks, tree labeling, guide pair, ranking.

1. Introduction

Self-stabilization [1, 2] is a useful property, enabling a distributed algorithm to withstand transient faults. A distributed algorithm is self-stabilizing if, after transient faults hit the system and place it in some arbitrary global state, the system recovers without external intervention in finite time.

An *ordered tree* $\mathcal{T}$ is a rooted tree, together with a *left-to-right order* on the children of each node. In this paper, we give two self-stabilizing distributed algorithms for ordered trees. Neither of the algorithms assumes knowledge of the size, n, of the tree, nor requires a known upper bound of n; although, as is customary in the literature, we assume that each process can store an integer in the range $1..n$, using $O(\log n)$ space.

[☆] This work has been partially supported by the ANR project *ARESA*2.

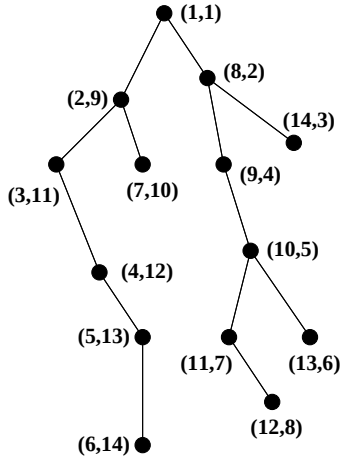


Figure 1: An ordered tree, labeled with guide pairs.

Our first algorithm, GUIDE, computes a *guide pair* for each process P, which we write as $P.guide = (P.pre\_ind, P.post\_ind)$, where $P.pre\_ind$ and $P.post\_ind$ are the rank of P in the *preorder* and *reverse postorder* traversal, respectively, of the ordered tree. Figure 1 shows an example of ordered tree labeled with guide pairs. The guide pairs provide a labeling scheme that can be used for various applications [3]. In this paper, we use these labels to navigate in the tree $\mathcal{T}$.

Our second algorithm, RANK, makes use of the guide pairs computed by GUIDE. The input of our second algorithm consists of a weight $P.weight$, of some ordered type, for each process P. RANK computes the *rank* of each process, *i.e.,* the node of smallest weight is given rank 1, the second smallest rank 2, and so forth.

1.1. Contributions

GUIDE has round complexity $O(h)$, where h is the height of $\mathcal{T}$. The round complexity of RANK is $O(n)$. The space complexity of GUIDE in each process P is $O(\delta_P \log n)$, where δ_P is the degree of P. RANK, which uses GUIDE as a subroutine, has space complexity $O(b+\delta_P \log n)$ in each process P, where b is the number of bits needed to store a weight. GUIDE and RANK are self-stabilizing. GUIDE is *silent*, that is, it eventually reaches a *final* configuration, where all actions are disabled. RANK correctly computes the rank of every process within $O(n)$ rounds, but is not silent. The ranks do not change once the system stabilizes. However, the algorithm repeatedly computes those ranks. If the weight do not change, the repeated computation of RANK will be transparent to any application that uses the output of RANK.

1.2. Related Work

The notion of guide pairs appeared first in [3], but the computation given in that paper is not self-stabilizing. To the best of our knowledge, there is

no previously published self-stabilizing algorithm for computing guide pairs; however, there exist several self-stabilizing algorithms for other kinds of labeling, *e.g.*, [4, 5]. Notice that our guide pair algorithm is a simple instantiation of the general approach given in [5], however we show the correctness of our algorithm assuming a distributed scheduler, while [5] assumes a central one.

The only previous self-stabilizing algorithm for the ranking problem is given in [6]. This algorithm works in rooted trees. Like ours, that algorithm is not silent. Unlike ours, it also assumes that each process has a unique identifier in the range $1..n$. The algorithm stabilizes in $O(nh)$ rounds using $O(\log n)$ space per process, where h is the height of the tree.

The ranking problem is related to the *sorting problem*. There are numerous self-stabilizing solutions for sorting in a tree, *e.g.*, [7, 8, 9].

1.3. Roadmap

In the next section, we describe the model we use throughout this paper. In Section 3, we give our self-stabilizing silent algorithm, GUIDE, for computing guide pairs. In Section 4, we give our self-stabilizing algorithm, RANK, for the ranking problem.

2. Preliminaries

Let $G = (V, E)$ be an undirected graph, where V is a set of nodes and E is a set of undirected edges linking nodes. Two nodes $P, Q \in V$ are said to be *neighbors* if $\{P, Q\} \in E$. The set of neighbors of P is denoted $N(P)$. The degree of P *i.e.,* $|N(P)|$, is denoted by δ_P. $G = (V, E)$ is a *tree* if it is connected and acyclic. A tree $\mathcal{T}$ can be *rooted* at some node, meaning that one of its nodes, *Root*, is distinguished to be the root. In a rooted tree $\mathcal{T}$, we denote by *P.par* the parent of node P in $\mathcal{T}$; if $P = Root$, then $P.par = P$; otherwise $P.par = Q$, where Q is the neighbor of P that is on the shortest path from P to *Root*. Let $Chldrn(P) = \{Q \in N(P) : Q.par = P\}$, the *children* of P in the tree $\mathcal{T}$. An *ordered tree* is a rooted tree $\mathcal{T}$, together with a local order, $\prec_P$, (which we call "left-to-right") on the children of every node. Let $P_1, P_2, \ldots P_m$ be the children of the root of $\mathcal{T}$ in the left-to-right order. We denote by $\mathcal{T}_i$ the subtree of $\mathcal{T}$ rooted at P_i.

In this paper, we assume the network is an ordered tree $\mathcal{T}$. We denote by $h(P)$ the height of process P in $\mathcal{T}$, *i.e.,* its distance to the root, and let $h = \max_{P \in V} h(P)$, the height of $\mathcal{T}$.

2.1. Computational Model

We consider the locally shared memory model, introduced by Dijkstra [1]. In this model, each process holds a finite set of shared *variables*. A process P can read its own variables and that of its neighbors, but can write only to its own variables. We also assume that, for any $Q \in N(P)$, P can determine whether $Q.par = P$. A *distributed algorithm* $\mathcal{A}$ is a collection of n *programs*, each one operating on a single process. The *program* of each process P in $\mathcal{A}$,

noted $\mathcal{A}(P)$, is a finite set of actions $\langle label \rangle :: \langle guard \rangle \mapsto \langle statement \rangle$. *Labels* are only used to identify actions in the discussion. The *guard* of an action in the program of a process P is a Boolean expression involving the variables of P and its neighbors. The *statement* of an action of P updates one or more variables of P. 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.

Let $\mathcal{A}$ be a distributed algorithm operating of a network G. The values of $\mathcal{A}$'s variables at some process P define the (*local*) *state* of P. A configuration of $\mathcal{A}$ in G is defined to be a vector consisting a local state of P for every P in G.

Let $\mapsto$ be the binary relation over configurations of $\mathcal{A}$ in G such that $\gamma \mapsto \gamma'$ if and only if it is possible for the configuration γ to change to configuration γ' in one step of $\mathcal{A}$. An *execution* of $\mathcal{A}$ is a maximal sequence of configurations $\varrho = \gamma_0\gamma_1 \ldots \gamma_i \ldots$ such that $\gamma_{i-1} \mapsto \gamma_i$ for all $i > 0$. The term "maximal" means that the execution is either infinite, or ends at a *final* configuration in which no action of any process is enabled. Each step $\gamma_i \mapsto \gamma_{i+1}$ consists of one or more *enabled* processes executing an enabled action. The evaluations of all guards and executions of all statements of those actions are presumed to take place in one atomic step. This model is called *composite atomicity* [2].

We say that a process P is *neutralized* in the step $\gamma_i \mapsto \gamma_{i+1}$ if P is enabled in γ_i and not enabled in γ_{i+1}, but does not execute any action at that step.

We assume that each step from a configuration to another is driven by a *scheduler*, also called a *daemon*. If one or more processes are enabled, the scheduler selects at least one of these enabled processes to execute an action. We assume that the scheduler is *weakly fair*, meaning that, every continuously enabled process is eventually selected by the scheduler. We say that an execution $\gamma_0\gamma_1 \ldots \gamma_i \ldots$ is *weakly fair* if every process that is enabled at any γ_i either executes an action or is neutralized at Step j for some $j > i$. Assuming the weakly fair scheduler is equivalent to the assumption that the scheduler never chooses an infinite execution that is not weakly fair.

To describe the time complexity of a distributed algorithm, we use the notion of *round*. The first *round* of an execution ϱ is defined to be the minimal prefix $\gamma_0 \cdots \gamma_i$ of ϱ in which every process that is enabled in the initial configuration either executes an action or becomes neutralized. The second round is the first round of the suffix of ϱ starting in γ_i, and so forth.

2.2. Self-stabilization and Silence

Suppose $\mathbb{P}$ is a predicate defined on all configurations of $\mathcal{A}$ on a network G. A distributed algorithm $\mathcal{A}$ is *self-stabilizing with respect to* $\mathbb{P}$ on a network G if

1. (Closure) If γ is a configuration of $\mathcal{A}$ on G and $\gamma \mapsto \gamma'$ is a step, then $\mathbb{P}(\gamma) \Rightarrow \mathbb{P}(\gamma')$.
2. (Convergence) Every weakly fair execution of $\mathcal{A}$ on G eventually satisfies $\mathbb{P}$.

Each algorithm given in this paper is associated with a specified *legitimacy predicate*, and we simply say that a configuration of that algorithm is *legitimate* if it is legitimate with respect to that predicate.

We say that an algorithm is *silent* [10] under the weakly fair daemon if every weakly fair execution of that algorithm is finite. In other words, starting from an arbitrary configuration, the network will eventually reach a configuration where no process is enabled.

2.3. Composition

We make use of *hierarchical collateral composition* [11], a variant of *collateral composition* [12]. When we collaterally compose two algorithms $\mathcal{A}$ and $\mathcal{B}$, they run concurrently and $\mathcal{B}$ uses some of the outputs of $\mathcal{A}$ as inputs. In the variant we use, we modify the code of $\mathcal{B}(P)$ (for every process P) so that P executes an action of $\mathcal{B}(P)$ only when it has no enabled action in $\mathcal{A}(P)$.

Let $\mathcal{A}$ and $\mathcal{B}$ be two (distributed) algorithms such that no variable written by $\mathcal{B}$ appears in $\mathcal{A}$. In the *hierarchical collateral composition* of $\mathcal{A}$ and $\mathcal{B}$, noted $\mathcal{B} \circ \mathcal{A}$, the (local) program of every process P, $\mathcal{B}(P) \circ \mathcal{A}(P)$, is defined as follows:

- $\mathcal{B}(P) \circ \mathcal{A}(P)$ contains all variables of $\mathcal{A}(P)$ and $\mathcal{B}(P)$.
- $\mathcal{B}(P) \circ \mathcal{A}(P)$ contains all actions of $\mathcal{A}(P)$.
- For every action $G_i \rightarrow S_i$ of $\mathcal{B}(P)$, $\mathcal{B}(P) \circ \mathcal{A}(P)$ contains the action $\neg D_P \wedge G_i \rightarrow S_i$, where D_P is the disjunction of all guards of actions in $\mathcal{A}(P)$.

The following sufficient condition is given in [11]:

Theorem 1. *The composite algorithm $\mathcal{B} \circ \mathcal{A}$ is self-stabilizing* w.r.t. *a predicate $\mathbb{P}$ on a network G assuming a weakly fair daemon if the following conditions hold:*

(a) *Algorithm $\mathcal{A}$ is silent and self-stabilizing with respect to a predicate $\mathbb{A}$ on G under the weakly fair daemon.*

(b) *Algorithm $\mathcal{B}$ is self-stabilizing with respect to $\mathbb{P}$ under the weakly fair daemon, provided predicate $\mathbb{A}$ holds and $\mathcal{A}$ is in a final configuration.*

Part (b) of Theorem 1 means that, if ϱ is a weakly fair execution of $\mathcal{B}$ such that the variables of $\mathcal{A}$ satisfy $\mathbb{A}$ and never change, then $\mathbb{P}$ eventually holds.

3. Computing Guide Pairs

3.1. Guide Pairs

Recall that we denote by $P_1, P_2, \ldots P_m$ the children of the root of $\mathcal{T}$ in the left-to-right order, and we denote by $\mathcal{T}_i$ be the subtree rooted at any P_i. The *preorder traversal* of $\mathcal{T}$ is defined, recursively, as follows:

1. Visit the root of $\mathcal{T}$.
2. For each i from 1 to m in increasing order, visit the nodes of $\mathcal{T}_i$ in *preorder*.

The reverse postorder traversal is defined similarly:

1. Visit the root of $\mathcal{T}$.
2. For i from m to 1 in decreasing order, visit the nodes of $\mathcal{T}_i$ in *reverse postorder*.

If a node P is the i^{th} node of $\mathcal{T}$ visited in a preorder traversal of $\mathcal{T}$, we say that the *preorder rank* of P is i. If a node P is the j^{th} node of $\mathcal{T}$ visited in a reverse postorder traversal of $\mathcal{T}$, we say that the *reverse postorder rank* of P is j. Write $pre\_ind(P)$ and $post\_ind(P)$ for the *preorder* rank and *reverse postorder* rank of P, respectively. We define the *guide pair* of P to be the ordered pair $guide(P) = (pre\_ind(P), post\_ind(P))$. Figure 1 (page 2) shows an ordered tree where each process is labeled with its guide pair.

We define a partial order on guide pairs: $(i, j) \leq (k, \ell)$ *iff* $i \leq k$ and $j \leq \ell$.

Remark 2. [Property 1 in [3]] *If P and Q are nodes of an ordered tree $\mathcal{T}$, then $guide(P) \leq guide(Q)$ if and only if P is an ancestor of Q.*

3.2. Algorithm GUIDE

Algorithm GUIDE is a hierarchical collateral composition of two algorithms: GUIDE = CGP $\circ$ COUNT, where both COUNT and CGP (for *Compute Guide Pairs*) use $P.par$ as input in the program of every process P.

3.2.1. Algorithm COUNT

COUNT is implemented as a bottom-up wave that computes the number of processes in each subtree. Each process P has only one variable: $P.subcount$, and one function: $Subcount(P) = 1 + \sum_{Q \in Chldrn(P)} Q.subcount$, and COUNT has only one action:

$$\texttt{SetCnt} \;::\; P.subcount \neq Subcount(P) \;\mapsto\; P.subcount \leftarrow Subcount(P)$$

The legitimacy predicate of COUNT is simply that $P.subcount = \|\mathcal{T}_P\|$ for all P, where $\|\mathcal{T}_P\|$ is the number of nodes in $\mathcal{T}_P$.

Lemma 3. COUNT *is self-stabilizing and silent, and converges within $h + 1$ rounds from an arbitrary initial configuration to a legitimate configuration.*

Proof: By induction on the height of $\mathcal{T}_P$. Within one round, $P.subcount = 1$ if P is a leaf of $\mathcal{T}$. Otherwise, If $\mathcal{T}_P$ has height t, then, by the inductive hypothesis, for every child Q of P, $Q.subcount = \|\mathcal{T}_Q\|$ if at least t rounds have elapsed, and thus $Subcount(P) = \|\mathcal{T}_P\|$. Within one more round, $P.subcount = \|\mathcal{T}_P\|$. □

3.2.2. Algorithm CGP

Using the values of *subcount* computed by COUNT, each process P evaluates in CGP for each of its children Q the number of processes before Q in the *preorder* and *reverse postorder* traversal of the tree $\mathcal{T}$, respectively (using Actions `SetChldPrePred` and `SetChldPostPred`, respectively). Then, reading these values from its parent, each process, except the root, can compute its guide pair (using Actions `SetPreInd` and `SetPostInd`). The guide pair of the root is $(1, 1)$ (see Actions `SetPreInd` and `SetPostInd` for the root).

Variables of CGP. The following array variable enables each non-root process to know its index in the local left-to-right order of its parent:

1. $P.child[k] \in N(P)$, for all $1 \leq k \leq \delta_P - 1$.
   This array is maintained by Action `SetChld`. For all $1 \leq k \leq \delta_P - 1$, $P.child[k]$ is set to the k^{th} child in P's local ordering of $Chldrn(P)$.

CGP uses the following additional variables:

2. $P.pre\_ind$, $P.post\_ind$, integers, which converge to the *preorder* and *reverse postorder* ranks of P, respectively. Thus, we will write $P.guide = (P.pre\_ind, P.post\_ind)$, the guide pair of P.
3. $P.chld\_pre\_pred[k]$, $P.chld\_post\_pred[k]$, integers, defined for all $1 \leq k \leq \delta_P - 1$:
   For all $1 \leq k \leq \delta_P - 1$, $P.chld\_pre\_pred[k]$ is set to the number of predecessors of the k^{th} child of P (that is, $P.child[k]$) in the *preorder* traversal of $\mathcal{T}$; and $P.chld\_post\_pred[k]$ is set to the number of predecessors of the k^{th} child of P in the *reverse postorder* traversal of $\mathcal{T}$.

   Hence, each process P computes its guide pair to be

   $$(P.par.chld\_pre\_pred[k] + 1, P.par.chld\_post\_pred[k] + 1)$$

   where k is the index of P in left-to-right order of its parent.

Functions of CGP. Using its variables and those of its neighbors, each process P can compute the following functions:

- $my\_order(P)$. (Only defined for non-root processes.)

  If there exists k, $1 \leq k \leq \delta_{P.par} - 1$, such that $P.par.child[k] = P$, then $my\_order(P)$ returns k. Otherwise, the values in $P.par.child$ have not stabilized yet and $my\_order(P)$ returns 1.

  Once the system has stabilized, $my\_order(P)$ returns the index of the non-root process P in the local left-to-right order of its parent.

- $Chld\_index(Q) = |\{Q' \in Chldrn(P) : Q' \prec_P Q\}| + 1$. $Chld\_index(Q)$ returns the index of the child Q of process P in the local left-to-right order of P.

- $Eval\_child(k)$ returns the local name of the k^{th} child of P. That is, $Eval\_child(k)$ returns $Q \in Chldrn(P)$ such that $Chld\_index(Q) = k$.

- $Eval\_child\_pre\_pred(k)$. If $k = 1$, then $Eval\_child\_pre\_pred(k)$ returns $P.pre\_ind$, else $Eval\_child\_pre\_pred(k)$ returns $P.chld\_pre\_pred[k-1] + P.child[k-1].subcount$.

  Once the system has stabilized, $Eval\_child\_pre\_pred(k)$ returns the number of predecessors of the k^{th} child of P in the *preorder* traversal of $\mathcal{T}$.

- *Eval_child_post_pred*(k). If $k = \delta_P - 1$, then *Eval_child_post_pred*(k) returns $P.post\_ind$, else *Eval_child_post_pred*(k) returns $P.chld\_post\_pred[k+1] + P.child[k+1].subcount$.

  Once the system has stabilized, *Eval_child_post_pred*(k) returns the number of predecessors of the k^{th} child of P in the reverse postorder traversal of $\mathcal{T}$.

Actions of CGP. Actions of CGP are given below. To simplify the presentation, we assume priorities on actions, and list them below in the order from the highest to the lowest priority. If several actions are enabled simultaneously at a process, only the one of the highest priority can be executed. In other words, the actual guard of any action "$L \;::\; G \mapsto S$" of process P is $\neg D \wedge G$, where D is the disjunction of the guards of all actions at P that appear earlier in the list.

For every process P**:**

`SetChld` :: $\exists k \in [1..\delta_P - 1], P.child[k] \neq Eval\_child(k)$
$\mapsto \; \forall k \in [1..\delta_P - 1], P.child[k] \leftarrow Eval\_child(k)$

`SetChldPrePred` :: $\exists k \in [1..\delta_P - 1], P.chld\_pre\_pred[k] \neq Eval\_child\_pre\_pred(k)$
$\mapsto \; \forall k \in [1..\delta_P - 1], P.chld\_pre\_pred[k] \leftarrow Eval\_child\_pre\_pred(k)$

`SetChldPostPred` :: $\exists k \in [1..\delta_P - 1], P.chld\_post\_pred[k] \neq Eval\_child\_post\_pred(k)$
$\mapsto \; \forall i \in [1..\delta_P - 1], P.chld\_post\_pred[k] \leftarrow Eval\_child\_post\_pred(k)$

For the root process *Root* **only:**

`SetPreInd` :: $Root.pre\_ind \neq 1 \quad \mapsto \; Root.pre\_ind \leftarrow 1$

`SetPostInd` :: $Root.post\_ind \neq 1 \quad \mapsto \; Root.post\_ind \leftarrow 1$

For every non-root process P **only:**

`SetPreInd` :: $P.pre\_ind \neq 1 + P.par.chld\_pre\_pred[my\_order(P)]$
$\mapsto \; P.pre\_ind \leftarrow 1 + P.par.chld\_pre\_pred[my\_order(P)]$

`SetPostInd` :: $P.post\_ind \neq 1 + P.par.chld\_post\_pred[my\_order(P)]$
$\mapsto \; P.post\_ind \leftarrow 1 + P.par.chld\_post\_pred[my\_order(P)]$

Overview of CGP. We now give an intuitive explanation of how CGP computes the values of $P.pre\_ind$ for all P. The values of $P.post\_ind$ are computed similarly.

Suppose that P is the i^{th} process visited in a *preorder* traversal of $\mathcal{T}$: i is the correct value of $P.pre\_ind$. CGP works by computing *Num_Preorder_Preds*(P), the number of predecessors of P in the preorder traversal, which is the correct value of $P.pre\_ind - 1$.

First, by definition, *Num_Preorder_Preds*$(Root) = 0$. Then, for every non-root process P, *Num_Preorder_Preds*(P) is computed by $P.par$ and stored in the variable $P.par.chld\_pre\_pred[k]$, where P is the k^{th} child of $P.par$ in left-to-right order. In order to compute these values for all its children, $P.par$ must have computed its own value of *pre_ind* as well as the sizes of all of its subtrees. If $k = 1$, then *Num_Preorder_Preds*$(P) = P.par.pre\_ind$, since $P.par$ is the immediate predecessor of its leftmost child in the *preorder* visitation. Thus, $P.par.chld\_pre\_pred[1] \leftarrow P.par.pre\_ind$. $P.par.chld\_pre\_pred[2]$ is obtained by adding the subtree size of the leftmost child of $P.par$ to $P.par.chld\_pre\_pred[1]$, since all members of that subtree are predecessors of the second child of $P.par$.

In general, the number of predecessors of P is equal to $P.par.pre\_ind$ plus the sum of the sizes of the leftmost $k-1$ subtrees of $P.par$. Similarly, the values of the array $P.par.chld\_post\_pred$ are computed from right to left. P then executes:

$$\begin{aligned} P.pre\_ind &\leftarrow P.par.chld\_pre\_pred[k]+1 \\ P.post\_ind &\leftarrow P.par.chld\_post\_pred[k]+1 \end{aligned}$$

Theorem 4. GUIDE *is self-stabilizing and silent, computes the guide pairs of all processes in $O(h)$ rounds from an arbitrary initial configuration, and works under the weakly fair scheduler.*

Proof: According to Theorem 1 and Lemma 3, to show that GUIDE is self-stabilizing, it is sufficient to show that CGP stabilizes from any silent legitimate configuration of COUNT.

In such a configuration, the value of $P.subcount$ is correct for all P. The variables of CGP are then computed in a top-down wave which takes $O(h)$ rounds. (We can prove this by induction on the height of processes in the tree, similar to the proof for COUNT.) Once a legitimate configuration is reached, no action is enabled.

Finally, the round convergence time of GUIDE is equal to the round convergence time of COUNT ($O(h)$ rounds) plus the number of rounds for CGP to reach a final configuration from any configuration where the values of all $P.subcount$ are correct ($O(h)$ rounds). □

4. Rank Ordering

In this section, we give an algorithm, RANK, that uses guide pairs to solve the *ranking problem* on an ordered tree, $\mathcal{T}$. We are given a value $P.weight$ for each process P in $\mathcal{T}$. For convenience, we can assume, in the discussion, that the weights are integers. The problem is to find the *rank* of each P. If $P_1, P_2, \ldots, P_n$ is the list of processes in $\mathcal{T}$ sorted by weight, then r is the rank of P_r.

Our algorithm RANK is a hierarchical collateral composition of two algorithms: RANK = CRK $\circ$ GUIDE. RANK computes the rank of each process P in $\mathcal{T}$, and sets the variable $P.rank$ to that value. RANK is self-stabilizing but not silent. It requires $O(n)$ rounds and $O(b+\delta_P \log n)$ space for each process P.

4.1. Overview of CRK

4.1.1. Flow of Packages

The key part of the algorithm CRK is the *flow of packages*. Each *package* is an ordered pair $x = (x.value, x.guide)$, where $x.value$ is its *value* and $x.guide$ is its *guide pair*. Moreover, for any two packages x and y, we say $x > y$ *iff* $x.value > y.value$.

Each package has a *home process* (the node from which the package is originally issued), although its *host* (location) can be at any process in the chain

between its home and the root. Each process can host up to two packages: one *up-package*, that is moving toward the root, and one *down-package*, that is moving back to its home. The guide pair of a package is the same as the guide pair of its home process, and its value is the weight of its home process if it is an up-package, and the rank that CRK will assign to its home process if it is a down-package.

Each process P initiates its flow of packages by creating an up-package whose value is $P.weight$. This up-package then moves to the root by forward copying. The flow of packages is organized so that packages with smaller weights reach the root before packages with larger weights, in a manner similar to the standard technique for maintaining min-heap order in a tree.

After the root copies an up-package from a child, it creates a down-package with the same home process as the up-package, but whose value is a number (a rank) in the range $1..n$. The root maintains a counter so that the first down-package it creates has value 1, the second value 2, and so forth. Each down-package then moves back to its home process by forward copying. When its home process copies a down-package, it assigns, or re-assigns, its rank to be the value of that package.

The purpose (in fact even the name) of the guide pair is now obvious. It is used to guide the down-package to its home process.

Since the root copies up-packages in weight order, it creates down-packages in that same order. The r^{th} down-package created by the root will carry rank r and will use the same guide pair as the r^{th} up-package copied by the root. Its home process will then be the process whose weight is the r^{th} smallest in $\mathcal{T}$.

RANK is not silent, but rather, endlessly repeats its computation. When the root detects that it has created all down-packages, it initiates a broadcast wave which resets the variables of CRK (except the rank and weight variables) and the computation of ranks starts over.

4.1.2. Redundant Packages

In our model of computation, if a variable of a process P is copied by a neighbor Q, it also remains at P. In the algorithm CRK, each process P can be home to at most one package, but we cannot avoid the existence of multiple copies of that package (up and/or down). We handle that problem by defining a package variable currently hosted by a process as being either *active* or *redundant*. A redundant package can be overwritten, but not an active package.

If x is an up-package currently hosted by some process Q which is not the root, then x is redundant if x has already been copied by $Q.par$. If x is an up-package currently hosted by the root, then x is redundant if the root has already created a down-package with the same guide pair as x. Any other up-package is active.

If x is a down-package hosted by some process Q which is not its home process, then Q is redundant if it has been copied by some child of Q. (The child that copies x must be the process whose subtree contains the home process of x.) If x is a down-package hosted by its home process P, then x is redundant if $P.rank$ is equal to the value of x, indicating that P has already copied its rank

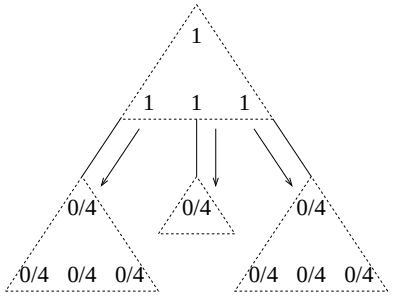
from x, or that $P.rank$ was correct before x arrived. Any other down-package is active.

4.1.3. Status Waves

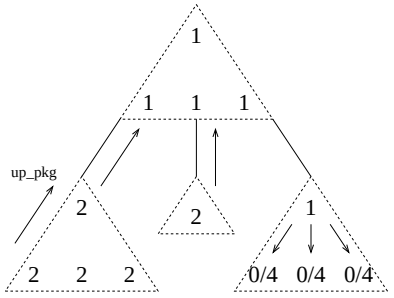
As it is typical for distributed algorithms which are self-stabilizing, but not silent, CRK endlessly repeats the calculation of the ranks of the processes in $\mathcal{T}$. We call one (complete) pass through this cycle of computations an *epoch*. At the end of each epoch, the variables of CRK at all processes, other than the variables for weight and rank, are reset for the next epoch. If an epoch has a "clean start", it will calculate the correct rank for each process. Subsequent epochs will simply recalculate the same value, and $P.rank$ will never change again. Thus, the ranks will eventually appear constant to the application.

On the other hand, in case of an arbitrary initial configuration, it is possible for incorrect values of rank to be calculated during the first epoch, but eventually a configuration will be reached where the next epoch will get a clean start.

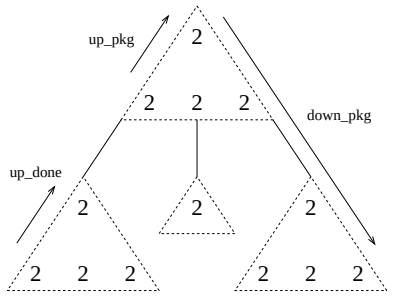
This system is controlled by the *status* variables of the processes. Status management is illustrated in Figure 2. At the beginning of an epoch, a broadcast wave starting from the root changes the status of every process from either 0 or 4 to 1, (Figure 2a), and all variables of CRK except rank and weight are set to their initial values. When this wave reaches the leaves of $\mathcal{T}$, a convergecast wave changes the status of all processes to 2 (Figure 2b). All computation of the ranking algorithm, as discussed above, takes place while processes have status 2. Once a process P detects that all processes in its subtree have created their own up-package and the subtree no longer contains any up-packages, it sets its Boolean variable $P.up\_done$ to TRUE (Figure 2c). After *Root* has created the last down-package, it also satisfies $Root.up\_done$ = TRUE, and consequently initiates a broadcast wave where the status of all processes change to 3 (Figure 2d). A process propagates the status 3 once its last down-package becomes redundant. The return convergecast wave then changes the status of all processes to 4 (Figure 2e), and when this wave reaches the root, the new epoch begins (Figure 2f).



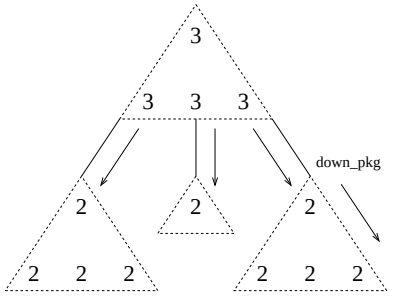
(a) Initial broadcast wave of status 1 resets variables.



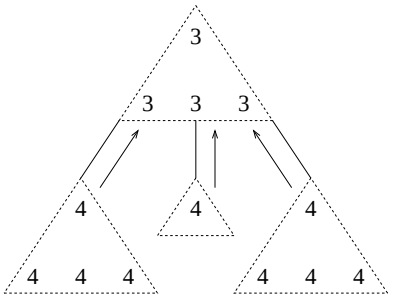
(b) Convergecast wave to status 2 allows for up-package propagation.



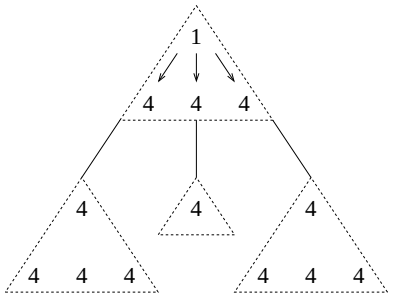
(c) Computation continues until $Root.up\_done = \text{TRUE}$.



(d) The last down-packages are pushed by broadcast wave of status 3.



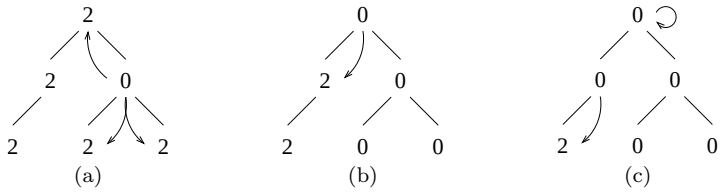
(e) The final convergecast wave of status 4 acknowledges receipt of all down-packages.



(f) Once *Root* has status 4, it starts a new epoch with a status 1 broadcast wave.

Figure 2: Status waves for a complete cycle of computations.

Status zero is used for error correction. If any process detects that the current epoch is erroneous, it changes its status to 0. Status 0 spreads down the tree, as well as up the tree unless it meets a process whose status is 1. If *Root.status* becomes 0 (and all its children have status 0 or 4), then *Root* initiates a status 1 broadcast wave starting a new epoch. However, we must prevent an endless cycle of 0 and 1 waves going up and down the tree respectively. We solve this problem by adding a special rule for the non-root processes. If $P.status = 0$ and $P.par.status = 1$, the status 0 wave cannot move up; instead, the status 0 wave moves only down, followed by the status 1 wave. This is illustrated in Figures 3 and 4.



(a) (b) (c)

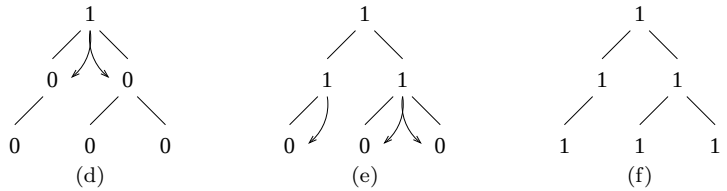


Figure 3: Error correction when root process gets status 0.

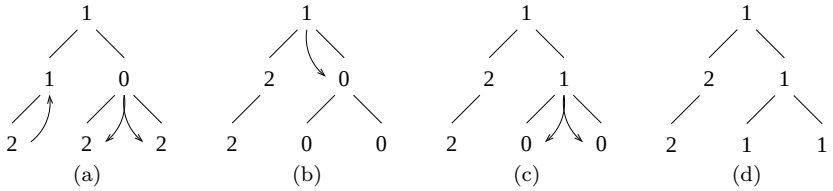


Figure 4: Error correction when *Root* already has status 1.

4.2. Formal Definition of CRK

4.2.1. Variables of CRK

Let P be any process. $P.par$, $P.guide$, and $P.weight$ are inputs of CRK. Then, the output of CRK is $P.rank$, an integer. To compute this output, P maintains the following additional variables:

1. $P.up\_pkg$ and $P.down\_pkg$ are respectively of package type (that is, a guide pair and an integer) or $\perp$ (undefined).
   If $P.up\_pkg$ (resp. $P.down\_pkg$) is defined, then its home process is some $Q \in \mathcal{T}_P$.
2. $P.started$, Boolean.
   This variable indicates that P has already generated its up-package during this epoch. ($P.up\_pkg$ may or may not still contain that up-package.)
3. $P.up\_done$, Boolean.
   This variable is TRUE if all processes in $\mathcal{T}_P$ have created their up-packages, and all such active up-packages have moved above P. Active up-packages whose home processes are in $\mathcal{T}_P$ could still exist at processes above P.
4. $P.status \in [0..4]$.
   Status variables are used to control the order of computation and to correct errors.

Finally, *Root* contains the following additional variable:

5. $Root.counter \in \mathbb{N}$.
   This incrementing integer variable assigns the rank to packages. It is initialized to be 0 every time a new epoch begins.

4.2.2. Predicates of CRK

The predicate *Clean_State*(P) below indicates that P is in a good ("clean") initial state.

$$Clean\_State(P) \equiv P.up\_pkg = \perp \wedge P.down\_pkg = \perp \wedge \neg P.started \wedge \neg P.up\_done$$

The following four predicates are used for error detection:

$$\begin{aligned}
Is_Consistent(P,g) &\equiv (g = P.guide) \vee (\exists Q \in Chldrn(P),\ g \geq Q.guide)\\
Guide_Error(P) &\equiv (P.up_pkg \neq \perp \wedge \neg Is_Consistent(P, P.up_pkg.guide)) \vee\\
&\quad (P.down_pkg \neq \perp \wedge \neg Is_Consistent(P, P.down_pkg.guide))\\
Status_Error(P) &\equiv (P.status \in \{1,3\} \wedge P.par.status \neq P.status) \vee\\
&\quad (P.status \in \{2,4\} \wedge (\exists Q \in Chldrn(P), Q.status \neq P.status)) \vee\\
&\quad (P.status \neq 0 \wedge P.par.status = 0) \vee\\
&\quad (P.status \notin \{0,1\} \wedge (\exists Q \in Chldrn(P), Q.status = 0))\\
Error(P) &\equiv Status_Error(P) \vee\\
&\quad (\neg Clean_State(P) \wedge P.status = 1) \vee\\
&\quad (Guide_Error(P) \wedge P.status = 2) \vee\\
&\quad (P.up_done \wedge \neg P.started \wedge P.status = 2) \vee\\
&\quad (P.up_done \wedge P.status = 2 \wedge (\exists Q \in Chldrn(P), \neg Q.up_done))
\end{aligned}$$

We say that a guide pair g is *consistent with* P if *Is_Consistent*(P,g) is TRUE. If *Is_Consistent*(P,g) is FALSE, g is the guide pair of no process in the subtree of P. *Guide_Error*(P) = TRUE means that P holds a package whose home is not in the subtree of P. The predicate *Status_Error*(P) indicates that P detects that its status is inconsistent with those of its neighbors. Status errors are always the result of arbitrary initialization; eventually, *Status_Error*(P) will become FALSE and will remain FALSE forever for all P. Finally, the predicate *Error*(P) detects error in the current wave.

The following four predicates are used for flow control:

$$\begin{aligned}
Up_Redundant(P) &\equiv (P \neq Root \wedge P.up_pkg \neq \perp \wedge P.par.up_pkg \neq \perp \wedge\\
&\quad P.par.up_pkg \geq P.up_pkg) \vee (P = Root \wedge P.up_pkg \neq \perp \wedge\\
&\quad P.down_pkg \neq \perp \wedge P.down_pkg.guide = P.up_pkg.guide)\\
Down_Ready(P) &\equiv P.down_pkg \neq \perp \Rightarrow ((P.down_pkg.guide \neq P.guide \wedge\\
&\quad (\exists Q \in Chldrn(P), Q.down_pkg = P.down_pkg)) \vee\\
&\quad (P.down_pkg.guide = P.guide \wedge P.rank = P.down_pkg.value))\\
Can_Start(P) &\equiv \neg P.started \wedge (P.up_pkg = \perp \vee Up_Redundant(P)) \wedge\\
&\quad (\forall Q \in Chldrn(P), (Q.up_done \vee\\
&\quad (\neg Up_Redundant(Q) \wedge Q.up_pkg > (P.weight, P.guide))))\\
Can_Copy_Up(P,Q) &\equiv Q \in Chldrn(P) \wedge (Q.up_pkg \neq \perp \wedge \neg Up_Redundant(Q)) \wedge\\
&\quad (P.up_pkg = \perp \vee Up_Redundant(P)) \wedge\\
&\quad (P.started \vee (P.weight, P.guide) > Q.up_pkg) \wedge\\
&\quad (\forall R \in Chldrn(P), (R.up_done \vee (R.up_pkg \neq \perp \wedge\\
&\quad \neg Up_Redundant(R) \wedge R.up_pkg \geq Q.up_pkg)))
\end{aligned}$$

P.up_pkg is redundant if *Up_Redundant*(P) is TRUE. *Down_Ready*(P) states that *P.down_pkg* is redundant or undefined, and thus P is permitted to create

or copy a new down-package. $Can\_Start(P)$ states that P can create its own package, that is, P can set $P.up\_pkg$ to $(P.weight, P.guide)$. $Can\_Copy\_Up(P, Q)$ states that P can copy $Q.up\_pkg$ to $P.up\_pkg$. We note that P can evaluate $Up\_Redundant(Q)$ for any $Q \in Chldrn(P)$.

4.2.3. Actions of CRK

Actions of CRK are given below. As earlier in the action table of CGP, they are listed in priority order.

For *Root* **only:**

`Err`	:: $Error(Root)$	$\mapsto$	$Root.status \leftarrow 0$
`NewEpoch`	:: $Root.status \in \{0,4\} \wedge$ $(\forall Q \in Chldrn(Root),$ $Q.status \in \{0,4\})$	$\mapsto$	$Root.status \leftarrow 1;\ counter \leftarrow 0$ $Root.up\_pkg \leftarrow \perp;\ Root.down\_pkg \leftarrow \perp$ $Root.started \leftarrow$ FALSE; $Root.up\_done \leftarrow$ FALSE
`ConvCast`	:: $Root.status = 1 \wedge$ $(\forall Q \in Chldrn(Root),$ $Q.status = 2)$	$\mapsto$	$Root.status \leftarrow 2$
`CreateUpPkg`	:: $Root.status = 2 \wedge$ $Can\_Start(Root)$	$\mapsto$	$Root.up\_pkg.value \leftarrow Root.weight;$ $Root.up\_pkg.guide \leftarrow Root.guide;$ $Root.started \leftarrow$ TRUE
`CopyUpPkg`	:: $Root.status = 2 \wedge$ $(\exists Q \in Chldrn(Root),$ $Can\_Copy\_Up(Root, Q))$	$\mapsto$	$Root.up\_pkg \leftarrow Q.up\_pkg,$ $Q = min_{\prec_{Root}}\{R \in Chldrn(Root),$ $Can\_Copy\_Up(Root, R)\}$
`EndUpPkg`	:: $Root.started \wedge$ $Up\_Redundant(Root) \wedge$ $(\forall Q \in Chldrn(Root),$ $Q.up\_done)$	$\mapsto$	$Root.up\_done \leftarrow$ TRUE
`CreateDownPkg`	:: $Down\_Ready(Root) \wedge$ $Root.up\_pkg \neq \perp \wedge$ $\neg Up\_Redundant(Root)$	$\mapsto$	$counter \leftarrow counter + 1;$ $Root.down\_pkg.value \leftarrow counter;$ $Root.down\_pkg.guide \leftarrow Root.up\_pkg.guide$
`SetRank`	:: $Root.down\_pkg \neq \perp \wedge$ $Root.down\_pkg.guide =$ $Root.guide \wedge$ $Root.down\_pkg.value \neq$ $Root.rank$	$\mapsto$	$Root.rank \leftarrow Root.down\_pkg.value$
`BroadCast`	:: $Root.status = 2 \wedge$ $Root.up\_done \wedge$ $Down\_Ready(Root)$	$\mapsto$	$Root.status \leftarrow 3$
`EndEpoch`	:: $Root.status = 3 \wedge$ $(\forall Q \in Chldrn(Root),$ $Q.status = 4)$	$\mapsto$	$Root.status \leftarrow 4$

For every non-root process P only:

`Err`	::	$Error(P)$	$\mapsto$	$P.status \leftarrow 0$
`NewEpoch`	::	$P.par.status = 1 \wedge$ $P.status \in \{0,4\} \wedge$ $(\forall Q \in Chldrn(P), Q.status \in \{0,4\})$	$\mapsto$	$P.status \leftarrow 1$; $P.up\_pkg \leftarrow \perp$; $P.down\_pkg \leftarrow \perp$; $P.started \leftarrow$ FALSE; $P.up\_done \leftarrow$ FALSE
`ConvCast`	::	$P.status = 1 \wedge$ $(\forall Q \in Chldrn(P), Q.status = 2)$	$\mapsto$	$P.status \leftarrow 2$
`CreateUpPkg`	::	$P.status = 2 \ \wedge \ Can\_Start(P)$	$\mapsto$	$P.up\_pkg.value \leftarrow P.weight$; $P.up\_pkg.guide \leftarrow P.guide$; $P.started \leftarrow$ TRUE
`CopyUpPkg`	::	$P.status = 2 \wedge$ $(\exists Q \in Chldrn(P), Can\_Copy\_Up(P,Q))$	$\mapsto$	$P.up\_pkg \leftarrow Q.up\_pkg$, $Q = \min_{\prec_P}\{R \in Chldrn(P),$ $Can\_Copy\_Up(P,R)\}$
`EndUpPkg`	::	$P.started \ \wedge \ Up\_Redundant(P) \wedge$ $(\forall Q \in Chldrn(P), Q.up\_done)$	$\mapsto$	$P.up\_done \leftarrow$ TRUE
`CopyDownPkg`	::	$Down\_Ready(P) \wedge$ $P.par.down\_pkg \neq \perp \wedge$ $P.par.down\_pkg \neq P.down\_pkg \wedge$ $Is\_Consistent(P, P.par.down\_pkg)$	$\mapsto$	$P.down\_pkg \leftarrow P.par.down\_pkg$
`SetRank`	::	$P.down\_pkg \neq \perp \wedge$ $P.down\_pkg.guide = P.guide \wedge$ $P.down\_pkg.value \neq P.rank$	$\mapsto$	$P.rank \leftarrow P.down\_pkg.value$
`BroadCast`	::	$P.par.status = 3 \ \wedge \ P.status = 2 \wedge$ $(\forall Q \in Chldrn(P), Q.status = 2) \wedge$ $Down\_Ready(P)$	$\mapsto$	$P.status \leftarrow 3$
`EndEpoch`	::	$P.status = 3 \wedge$ $(\forall Q \in Chldrn(P), Q.status = 4)$	$\mapsto$	$P.status \leftarrow 4$

The actions above achieve three tasks. They are (1) error correction, (2) control of epochs, and (3) rank computation (using the flow of packages).

Error Correction. Action `Err` performs the error correction. If one process detects any inconsistency among its state and that of its neighbors, it initiates a reset of the network by changing its status to 0.

Epochs. We now describe what happens during one epoch. In this description, we assume that the epoch contains no initialization errors. (As mentioned above, if any process detects such an error, the epoch is aborted, and a new, error-free, epoch begins.)

The new epoch starts when *Root* executes Action `NewEpoch`. If *Root.status* is either 0 or 4, and every child of *Root* has status 0 or 4, then *Root* broadcasts a status 1 wave and resets to a clean state.

When the status 1 wave reaches the leaves, all processes execute Action `ConvCast` in a convergecast wave, changing status to 2, so that rank computation can begin.

When *Root* detects that there are no more up-packages in the tree, and it has already sent every down-package, it initializes a status 3 broadcast wave

by executing Action `BroadCast`. Note that there could still be active down-packages below *Root*, but there could not be any active up-packages. Thus, *Root* is finished with its task for the current epoch. A non-root process P propagates the status 3 wave by Action `BroadCast` after sending all its down-packages. There could still be active down-packages below P, but no active up-packages. Since $P.par.status = 3$ and the down-package at P is redundant, P knows that its job for this epoch is done, and so changes its status to 3.

Once the status 3 wave reaches the leaves, all process execute Action `EndEpoch` in a convergecast status 4 wave. When that wave reaches *Root*, the current epoch is done, and *Root* initiates a new epoch.

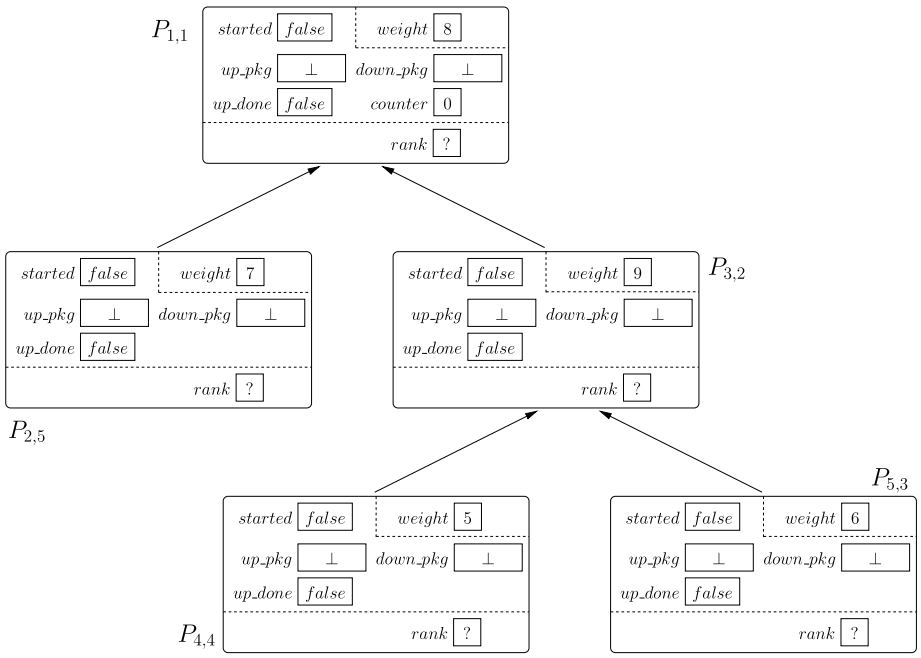
Rank Computation. The computation of the ranks is bottom-up, and starts when the convergecast status 2 wave starts at the leaves. The flow of up-packages is organized using `CreateUpPkg` and `CopyUpPkg`, that is, a process either inserts its own package in the flow or copies some package coming from a child in such a way to ensure that packages are moved up in ascending order of weight. Once a process P has detected that $\mathcal{T}_P$ has no active up-package, it sets $P.up\_done$ to TRUE by Action `EndUpPkg`. *Root* initializes the broadcast of the status 3 wave only after $Root.up\_done$ changes to TRUE.

When *Root* receives a new up-package, that is, $Root.up\_pkg$ becomes active, if $Root.down\_pkg$ is available (that is, it is either $\perp$ or redundant), *Root* is enabled to create a new down-package by executing `CreateDownPkg`. If $counter = r$, then $Root.up\_pkg$ is the r^{th} up-package copied or created by *Root*, *i.e.,* its weight is the r^{th} smallest weight in the network; r will then become the value of the down-package.

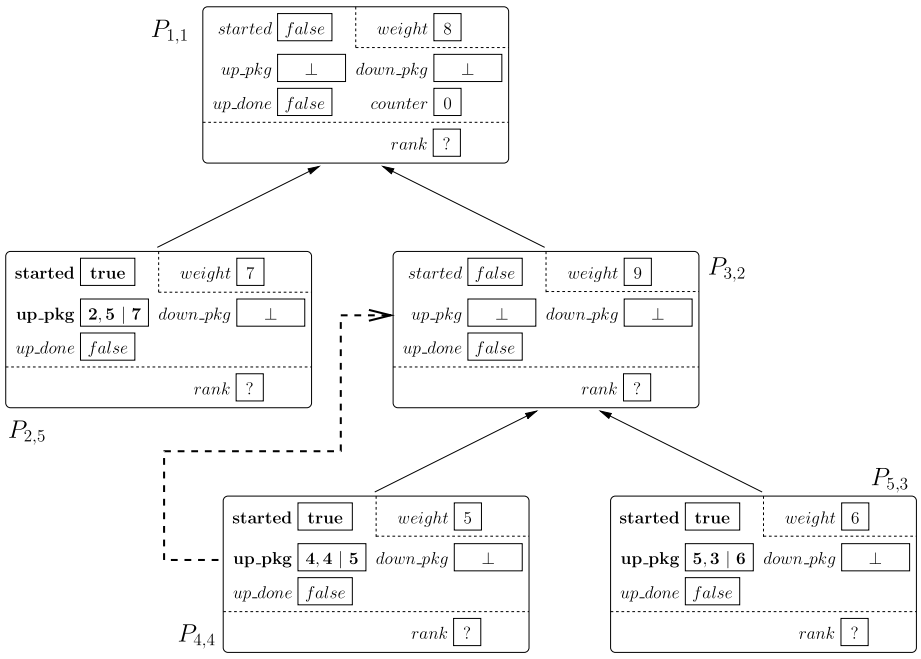
The new active down-package is propagated to its home process by forward copying, guided by its guide pair, using Action `CopyDownPkg`. When it reaches its home process P, the value field of that package contains the correct value of the rank of P. P updates $P.rank$ using Action `SetRank`, if necessary.

Figure 5 depicts a synchronous execution of a rank computation. For every process P, we show its inputs (processes are subscripted with their guide pair and their weight is given upper right), some of its computation variables (in the middle: up-package, down-package, *up_done* and *started* flags and root-counter) and its output (at the bottom: rank). At each step, when the value of a variable changes, we write the new value in bold. Dashed arrows show the next moves of a up- or down-package.

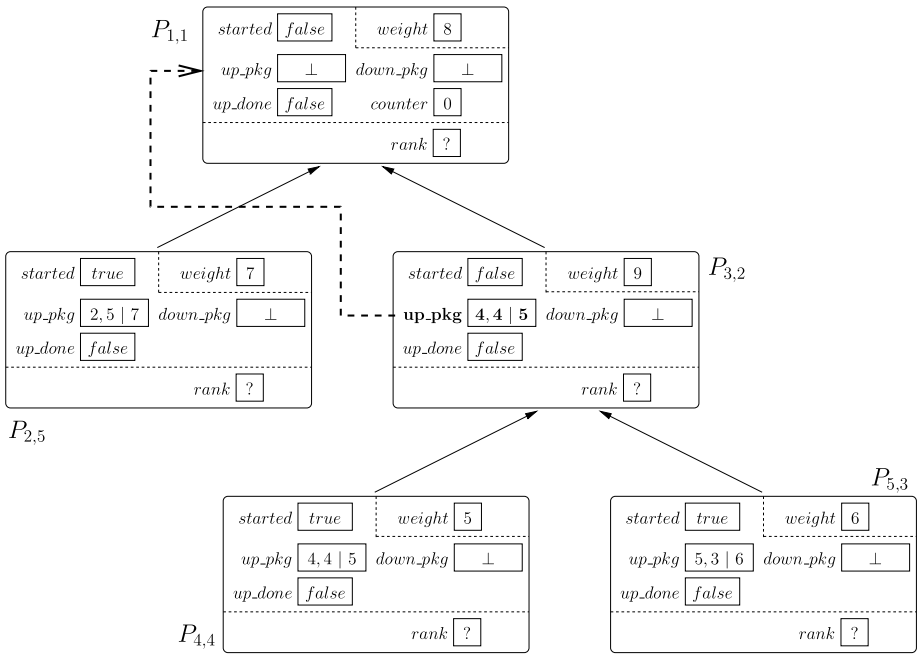
The example starts in a configuration where every computation variable has been reset by Action `NewEpoch` (Figure 5a). The output variables *rank* hold arbitrary values, denoted by "?." In Figure 5b, every leaf creates its own up-package with its guide pair and its weight. The up-packages are then routed, in weight order, up to the root, as shown in Figures 5c and 5d. In Figure 5e, the root process $Root = P_{1,1}$ increments its counter to 1 and creates the first down-package of the current epoch: the smallest weight is 5 and is held by the process labeled by the guide pair $(4, 4)$. This down-package is routed down to $P_{4,4}$, thanks to guide pairs, as shown in Figures 5f and 5g. Finally, in Figure 5h, $P_{4,4}$ assigned its own rank.


$P_{1,1}$
started false
weight 8
up_pkg ⊥
down_pkg ⊥
up_done false
counter 0
rank ?
started false
weight 7
up_pkg ⊥
down_pkg ⊥
up_done false
rank ?
$P_{2,5}$
started false
weight 9
up_pkg ⊥
down_pkg ⊥
up_done false
rank ?
$P_{3,2}$
started false
weight 5
up_pkg ⊥
down_pkg ⊥
up_done false
rank ?
$P_{4,4}$
started false
weight 6
up_pkg ⊥
down_pkg ⊥
up_done false
rank ?
$P_{5,3}$

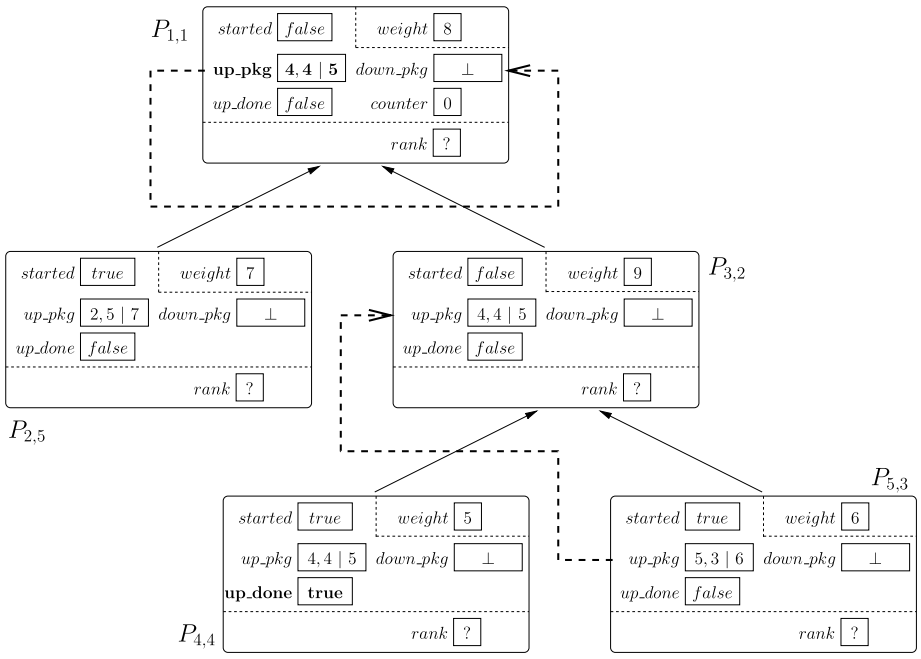
(a)


$P_{1,1}$
started false
weight 8
up_pkg ⊥
down_pkg ⊥
up_done false
counter 0
rank ?
started true
weight 7
up_pkg 2, 5 | 7
down_pkg ⊥
up_done false
rank ?
$P_{2,5}$
started false
weight 9
up_pkg ⊥
down_pkg ⊥
up_done false
rank ?
$P_{3,2}$
started true
weight 5
up_pkg 4, 4 | 5
down_pkg ⊥
up_done false
rank ?
$P_{4,4}$
started true
weight 6
up_pkg 5, 3 | 6
down_pkg ⊥
up_done false
rank ?
$P_{5,3}$

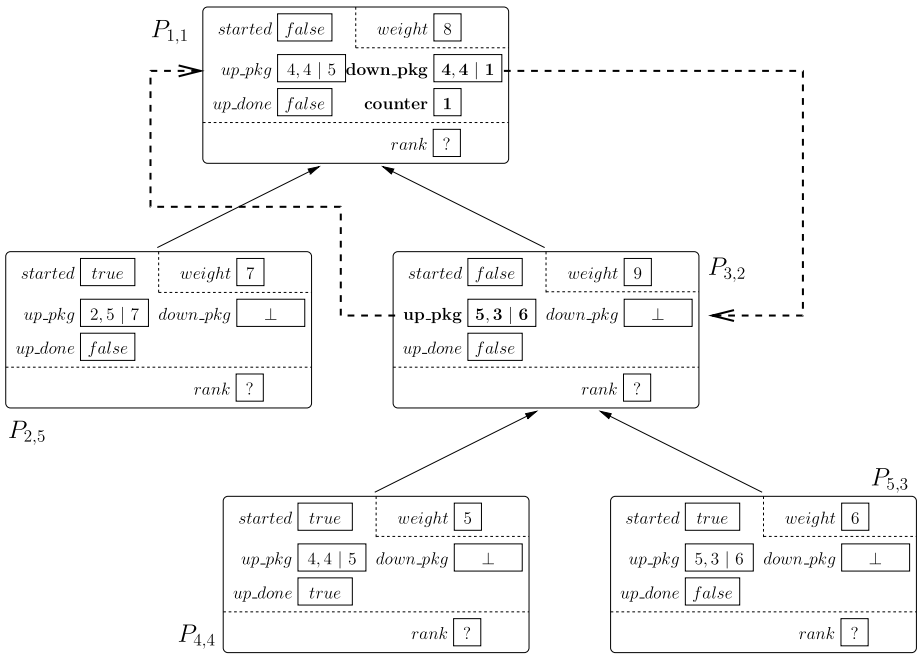
(b)


$P_{1,1}$
started false
weight 8
up_pkg ⊥
down_pkg ⊥
up_done false
counter 0
rank ?
started true
weight 7
up_pkg 2, 5 | 7
down_pkg ⊥
up_done false
rank ?
$P_{2,5}$
started false
weight 9
$P_{3,2}$
up_pkg 4, 4 | 5
down_pkg ⊥
up_done false
rank ?
$P_{5,3}$
started true
weight 5
up_pkg 4, 4 | 5
down_pkg ⊥
up_done false
rank ?
$P_{4,4}$
started true
weight 6
up_pkg 5, 3 | 6
down_pkg ⊥
up_done false
rank ?

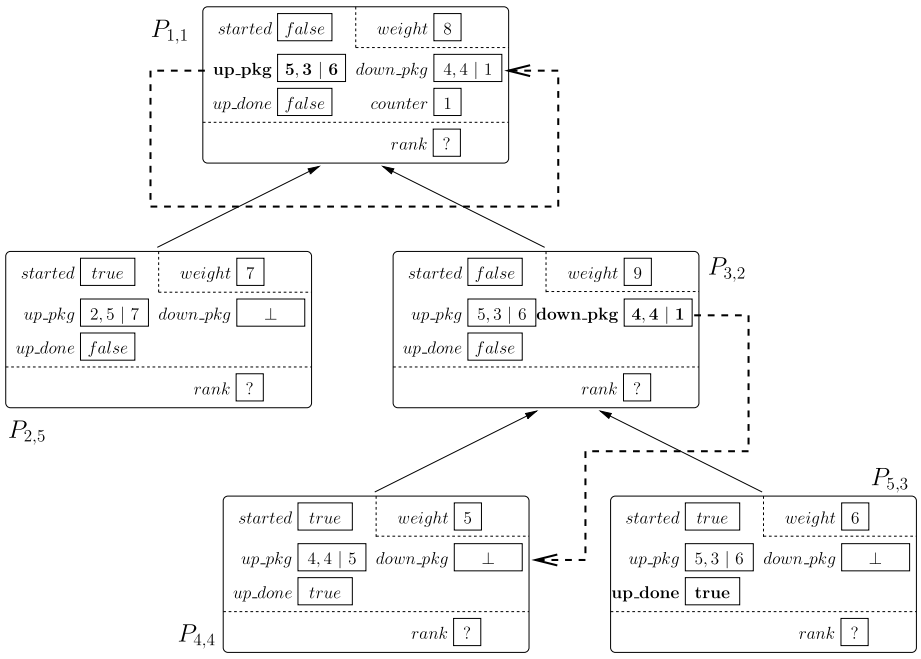
(c)


$P_{1,1}$
started false
weight 8
up_pkg 4, 4 | 5
down_pkg ⊥
up_done false
counter 0
rank ?
started true
weight 7
up_pkg 2, 5 | 7
down_pkg ⊥
up_done false
rank ?
$P_{2,5}$
started false
weight 9
$P_{3,2}$
up_pkg 4, 4 | 5
down_pkg ⊥
up_done false
rank ?
$P_{5,3}$
started true
weight 5
up_pkg 4, 4 | 5
down_pkg ⊥
up_done true
rank ?
$P_{4,4}$
started true
weight 6
up_pkg 5, 3 | 6
down_pkg ⊥
up_done false
rank ?

(d)


$P_{1,1}$
started false
weight 8
up_pkg 4, 4 | 5
down_pkg 4, 4 | 1
up_done false
counter 1
rank ?
started true
weight 7
up_pkg 2, 5 | 7
down_pkg ⊥
up_done false
rank ?
$P_{2,5}$
started false
weight 9
$P_{3,2}$
up_pkg 5, 3 | 6
down_pkg ⊥
up_done false
rank ?
$P_{5,3}$
started true
weight 5
up_pkg 4, 4 | 5
down_pkg ⊥
up_done true
rank ?
$P_{4,4}$
started true
weight 6
up_pkg 5, 3 | 6
down_pkg ⊥
up_done false
rank ?

(e)


$P_{1,1}$
started false
weight 8
up_pkg 5, 3 | 6
down_pkg 4, 4 | 1
up_done false
counter 1
rank ?
started true
weight 7
up_pkg 2, 5 | 7
down_pkg ⊥
up_done false
rank ?
$P_{2,5}$
started false
weight 9
$P_{3,2}$
up_pkg 5, 3 | 6
down_pkg 4, 4 | 1
up_done false
rank ?
$P_{5,3}$
started true
weight 5
up_pkg 4, 4 | 5
down_pkg ⊥
up_done true
rank ?
$P_{4,4}$
started true
weight 6
up_pkg 5, 3 | 6
down_pkg ⊥
up_done true
rank ?

(f)

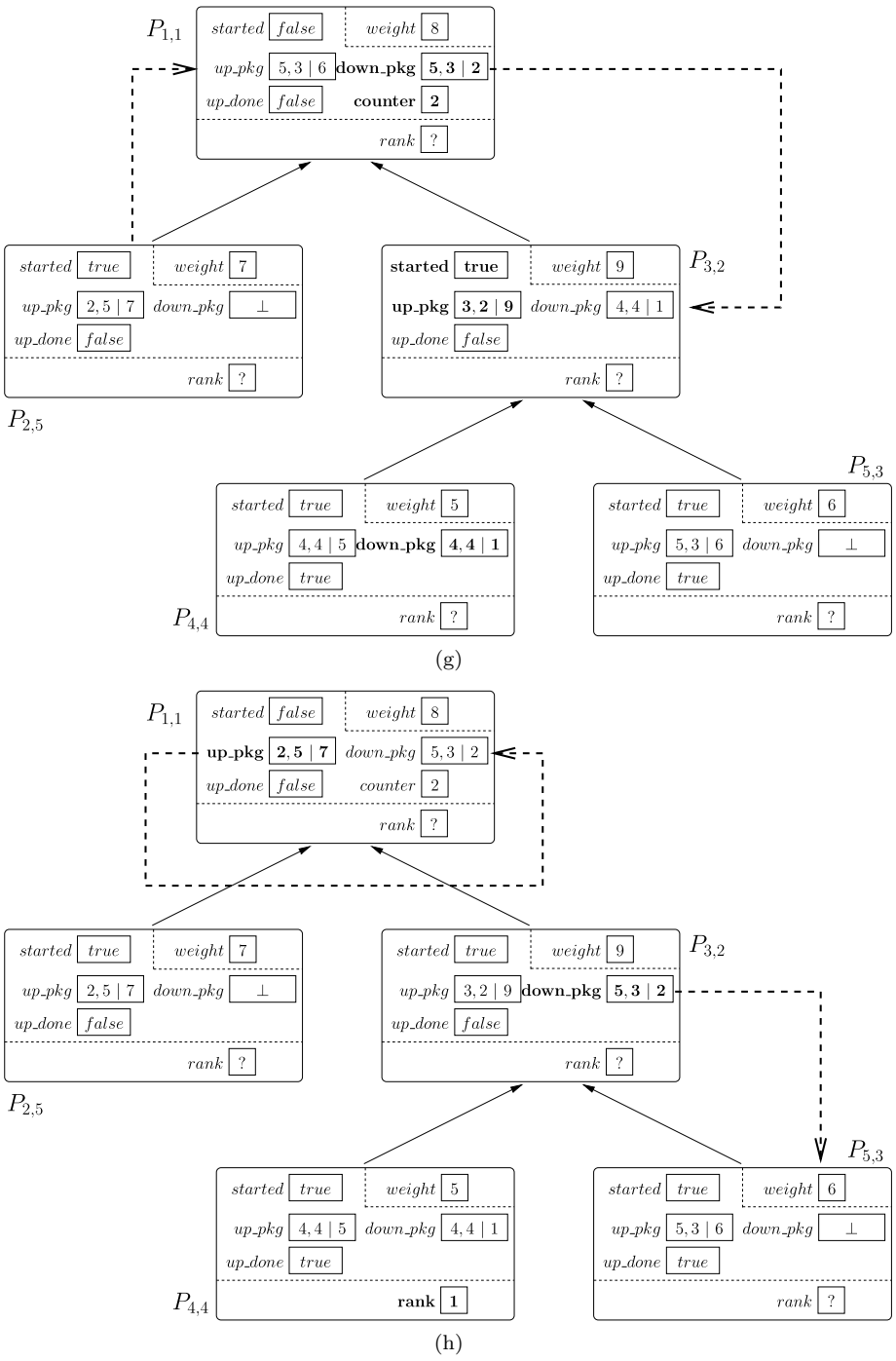


Figure 5: Example of an execution until the first rank is assigned.

4.2.4. Correctness of CRK

By Theorem 1, to show the correctness of RANK, it suffices to show that the variables of CRK stabilize to their correct values, starting from any silent legitimate configuration of GUIDE. Let γ be such a configuration. The first part of the proof deals with error correction.

We say that a process P is *inconsistent* if $P.status = 2$, $P.up\_done$, and there is some $Q \in Chldrn(P)$ such that $Q.up\_done =$ FALSE.

Lemma 5. *If at least one round has elapsed after configuration γ, the following conditions hold for every process P:*

(a) $\neg Clean\_State(P) \wedge (P.status = 1)$ *is* FALSE.

(b) $P.up\_done \wedge \neg P.started \wedge (P.status = 2)$ *is* FALSE.

(c) P *is not inconsistent;* i.e., *if* $P.status = 2$ *and* $P.up\_done =$ TRUE, *then* $Q.up\_done =$ TRUE *for all* $Q \in Chldrn(P)$.

Proof: Let consider the three conditions separately.

(a) If $Clean\_State(P) =$ FALSE and $P.status = 1$, then P is enabled to execute Action `Err`. Moreover, this condition only deals with local variables of P. So, Action `Err` is continuously enabled, and P executes $P.status \leftarrow 0$ in at most one round. Then, $\neg Clean\_State(P) \wedge (P.status = 1)$ is FALSE.

Assume $\neg Clean\_State(P) \wedge (P.status = 1)$ is FALSE. Then, if $P.status = 1$, P cannot modify its other variables before changing its status. Moreover, every time $P.status$ is reset to 1, the other variables are reset to a *clean* state (see Action `NewEpoch`). So, $\neg Clean\_State(P) \wedge (P.status = 1)$ remains FALSE forever.

(b) If $P.up\_done =$ TRUE, $P.started =$ FALSE, and $P.status = 2$, then P is enabled to execute Action `Err`. Moreover, this condition only deals with local variables of P. So, Action `Err` is continuously enabled, and P executes $P.status \leftarrow 0$ in at most one round. Then, $P.up\_done \wedge \neg P.started \wedge (P.status = 2)$ is FALSE.

Assume $P.up\_done \wedge \neg P.started \wedge (P.status = 2)$ is FALSE. Then, P always sets $P.up\_done$ and $P.started$ to FALSE together in Action `NewEpoch`. Moreover, P sets $P.up\_done$ to TRUE only if $P.started$ holds (see Action `EndUpPkg`). So, $P.up\_done \wedge \neg P.started \wedge (P.status = 2)$ remains FALSE forever.

(c) Assume P is inconsistent. Then, in one round, either every $Q \in Chldrn(P)$ satisfies $Q.up\_done = true$ or P executes Action `Err`. In both cases, P is no more inconsistent.

Assume P is not inconsistent. Then, P sets $P.up\_done$ to TRUE, by executing Action `EndUpPkg`, only when every $Q.up\_done =$ TRUE for all

$Q \in Chldrn(P)$. Moreover, any $Q \in Chldrn(P)$ sets $Q.up\_done$ to FALSE, by executing Action `NewEpoch`, only when $P.up\_done =$ FALSE. Thus, P cannot later become inconsistent.

□

Lemma 6. *If at least one round has elapsed after configuration γ, and if $Status\_Error(P) =$ TRUE, then one of the following conditions holds:*

- *$P \neq Root$ and $P.par.status = 0$.*
- *There is some $Q \in Chldrn(P)$ such that $Q.status = 0$.*

Proof: First, values 1 and 3 are propagated in the tree by broadcast waves. Then, values 2 and 4 are propagated in the tree by convergecast waves. So, by definition of $Status\_Error(P)$, if $Status\_Error(P) =$ FALSE at some point, then $Status\_Error(P)$ will become TRUE only after some neighbor of P switches its status to 0. Finally, by the definition of Action `Err`, P cannot satisfy $Status\_Error(P) =$ TRUE during one round without changing its status to 0. □

Lemma 7. *If a process with status 0 holds an active package, this package remains blocked until it is removed or cleaned.*

Proof: If a process P has status 0, then no other process can copy its up or down packages because each of its neighbors either has status 0, is its parent and has status 1, or is enabled to execute Action `Err`, the action with the highest priority. The next time P changes its status by executing Action `NewEpoch`, its state will become clean. □

Lemma 8. *Within $O(n)$ rounds from γ, if process P contains an active package such that there is no process in its subtree which is the home process of that package, then $P.status = 0$.*

Proof: Consider any configuration γ' after one round from γ. Consider an active package x in γ' at any process P such that there is no process in the subtree of P that is the home process of that package.

Assume that there is an ancestor of P with status 0, or a process in the subtree of P with status 0. Then, in at most h rounds, any process that holds x as an active package has status 0 by Action `Err` (remember that processes with status 1 do not hold any package, by Lemma 5), and by Lemma 7, x cannot be copied anymore, so we are done.

Assume that no ancestor and no descendant of P have status 0. We have four cases, depending on the status of P.

(a) $P.status = 4$. Assume that there is an ancestor Q of P whose status is 1. By Lemma 6 and the definition of *Status_Error*, all descendants of P have status 4, and for every ancestor R of P, we have $R.status \in \{1, 4\}$ and $(R.status = 1) \Rightarrow (R = Root) \vee (R.par.status = 1)$. Thus, in at most h rounds, the subtree of P has been reset to a clean state by Action `NewEpoch`, and we are done.

Assume that there is no ancestor Q of P such that $Q.status = 1$. Then, by Lemma 6 and the definition of *Status_Error*, all descendants of P have status 4, and for every ancestor R of P we have $R.status \in \{3, 4\}$ and $(R.status = 3) \Rightarrow (R = Root) \vee (R.par.status = 3)$. Thus, in at most h rounds, all ancestors of P will change to status 4 by executing Action `EndEpoch`, and we reduce to the previous case.

(b) $P.status = 3$. If there is a process that has status 4, we reduce to the previous case, by Lemma 6 and the definition of *Status_Error*.

Otherwise, every process of the tree has status 2 or 3, and if a process has status 3, then either it is *Root*, or its parent also has status 3, by Lemma 6 and the definition of *Status_Error*. In this case, x can only be copied down in the tree (and only if it is a down package). In $O(n)$ rounds, one of the following conditions will hold.

(i) x becomes an active package of a node Q such that $Guide\_Error(Q) \wedge (Q.status = 2)$. (In the worst case Q is a leaf.) The children of Q cannot copy x, and after one additional round, Q has status 0, and x cannot be copied anymore, by Lemma 7, so we are done.

(ii) The broadcast wave of status 3 reaches the leaves of the tree, and in at most h additional rounds, after the convergecast of the status 4 wave, we have Case (a).

(c) $P.status = 2$. If there is a process Q such that $Q.status = 3$, then $Root.status = 3$ by Lemma 6 and the definition of *Status_Error*, and we reduce to Case (b).

Otherwise, by Lemma 6, every process has status 1 or 2, and if a process has status 1, either it is *Root* or its parent has status 1. By executing Action `ConvCast`, all processes of $\mathcal{T}$ have status 2 within at most h rounds.

- If x is an up-package, it can only be copied up the tree. Either P satisfies $Guide\_Error(P) \wedge (P.status = 2)$, its parent cannot copy x, after one round P has status 0, and x cannot be copied any more (by Lemma 7), so we are done; or in $O(n)$ rounds, x becomes a down package at the *Root*, which has status 2, and is no longer an active up-package at any process.
- If x is a down-package, it can only be copied down in the tree. After $O(n)$ rounds, the host Q of x satisfies $Guide\_Error(Q) \wedge (Q.status = 2)$. (In the worst case Q is a leaf.) The children of Q cannot copy x. After one additional round, Q has status 0, and, by Lemma 7, x cannot be copied anymore; hence we are done.

(d) $P.status = 1$. By Lemma 5, P does not hold any package, so this case is contradictory.

□

Lemma 9. *Within $O(n)$ rounds from γ, if a process P contains a package, then there is a process in its subtree which is the home process of that package.*

Proof: By Lemmas 7 and 8, after $O(n)$ rounds, every process P holding an active package that does not have its home in the subtree of P satisfies $P.status = 0$ and no process copies this package.

The status 0 wave is propagated in $O(h)$ rounds, by Action `Err`, up the tree until reaching the root, or a process with status 1 all of whose ancestors also have status 1, causes all processes in the subtree $\mathcal{T}_P$ to change their status to 1 within $O(h)$ rounds by executing Action `NewEpoch`.

Hence, within $O(n)$ rounds, all inconsistent active packages will be removed from the tree, by Lemma 6. □

By Lemmas 5, 6, and 9, within $O(n)$ rounds from γ, $Error(P)$ is FALSE forever for each process P. There may still exist processes with status 0, but in that case, by the definition of *Error*, and for any process P, we have $P.status\{0,1,4\}$, $(P.status = 0) \Rightarrow (P = Root) \vee (P.par.status \in \{0,1\})$, $(P.status = 1) \Rightarrow (P = Root) \vee (P.par.status = 1)$, and $(P.status = 4) \Rightarrow (P \neq Root) \wedge (P.par.status \in \{1,4\})$. Hence, at the end of the status 1 broadcast wave, which takes at most $O(h)$ rounds, no process will have status 0. Thus, we have the following lemma:

Lemma 10. *Within $O(n)$ rounds after configuration γ, $Error(P)$ is* FALSE *and $P.status \in \{1,2,3,4\}$ forever, for each process P.*

From Lemma 10, we can deduce that the following invariant holds within $O(n)$ rounds after γ for all P.

1. $Error(P)$ is FALSE and $P.status \in \{1,2,3,4\}$.

   That is, all initial errors will eventually be corrected.

2. If $P.status \in \{1,3\}$, then either $P = Root$ or $P.par.status = P.status$.

3. If $P.status \in \{2,4\}$, then $Q.status = P.status$ for all $Q \in Chldrn(P)$.

We now show that, starting from any configuration where all previous invariants hold, infinitely many complete epochs are executed, and each of those epochs takes $O(n)$ rounds.

- If $Root.status = 4$, then all processes have status 4 and *Root* initiates a status 1 broadcast wave by executing Action `NewEpoch`.

- If $Root.status = 1$, then all processes P have either status 1 or 4. Moreover, $(P.status = 1) \Rightarrow (P = Root) \vee (P.par.status = 1)$. Thus, the status 1 broadcast wave reaches all processes in at most h rounds.

- After the status 1 wave reaches the leaves, the status 2 convergecast wave is initiated by the leaves by execution of Action `ConvCast`, and moves to *Root* in at most h rounds.

- Once $Root.status = 2$, all processes have status 2. The flow of packages starts in parallel at processes of status 2.

- By Claim 1, for every process P, if $P.status = 2$ and $P.up\_done$, every process Q in $\mathcal{T}_P$ subtree satisfies $Q.up\_done$. Moreover, $\neg P.started \Rightarrow \neg P.up\_done$. By executing Action `CreateUpPkg`, the deepest node P satisfying $P.status = 2$ and $\neg P.started$ eventually sets $P.started$ to TRUE and initiates its own up-package. The up-packages go up in the tree in weight order. Every process P satisfies $P.up\_done$ after $O(n)$ rounds.

- When each process P satisfies $P.up\_done$, *Root* eventually satisfies $Down\_Ready(Root)$. Then, *Root* initiates the status 3 broadcast wave by executing Action `BroadCast`.

- When $Root.status = 3$, all processes P have either status 2 or 3. Moreover, $(P.status = 3) \Rightarrow (P = Root) \vee (P.par.status = 3)$. So, status 3 is broadcast to the whole tree by Action `BroadCast`. As each node must wait for its down-package to become redundant before switching to status 3, this phase is takes $O(n)$ rounds.

- Finally, once the status 3 wave reaches a leaf, the status 4 convergecast wave is initiated. That wave is completed within at most h rounds. *Root* eventually has status 4, again.

Consider now any epoch that starts from a configuration, where all previous invariants (1-3) hold. We define $\mathcal{S} = \{Q : Q.status \in \{1, 2, 3\}\}$. We call $\mathcal{S}$ the *active portion* of $\mathcal{T}$. The following invariants hold for all $P \in \mathcal{S}$.

4. If $P.status = 1$, then $P.started$ and $P.up\_done$ are FALSE, and $P.up\_pkg = P.down\_pkg = \perp$.

   If the status of P is 1, then P has initialized its variables and has not yet begun the calculations of the epoch.

   Proof: By Claim 1, and definitions of $Error(P)$ and $Clean\_State(P)$. □

5. If $P.up\_done$ then $P.started$, and $Q.up\_done$ for all $Q \in Chldrn(P)$.

   If there is no active up-package in $\mathcal{T}_P$, then there is no active up-package in $\mathcal{T}_Q$ for any child Q. Furthermore, the package whose home is P has already been created and copied up.

   Proof: $P.up\_done$ is initialized to FALSE for all processes P during the broadcast wave of status 1 (Claim 4). Then, all $P.up\_done$ are set to TRUE in a bottom up fashion by Action `EndUpPkg`. □

6. *P.up_done* if and only if there is no active up-package in $\mathcal{T}_P$.

   Proof: *P.up_done* is initialized to FALSE for all processes P during the broadcast wave of status 1 (Claim 4). Then, all *P.up_done* are set to TRUE in a bottom up fashion by Action `EndUpPkg`. We can verify this claim by induction. □

7. If P hosts an active up-package, there is some process $Q \in \mathcal{T}_P \cap \mathcal{S}$ that is the home process of that package.

   Proof: If there is no process $Q \in \mathcal{T}_P$ that is the home process of that package, then in $O(n)$ rounds, some process R satisfies $R.status = 0$ by Lemma 8, a contradiction to Claim 1.

   Assume that $Q \notin \mathcal{S}$, that is, $Q.status = 4$. Then, $Root.status \in \{3, 4\}$ by Claims 1-3. Before satisfying $Root.status \in \{3, 4\}$, *Root* has changed its status from 1 to 2 and from 2 to 3. But, *Root* changes its status to 3 only if *Root.up_done* (see Action `BroadCast`). In this case, there is no active up-package in $\mathcal{T}$ by Claim 6, a contradiction. □

8. If *P.started* is FALSE, then there is no active package in $\mathcal{S}$ whose home process is P.

   Proof: P resets *P.started* to FALSE during the status 1 broadcast wave (Claim 4). Moreover, when P receives status 1 broadcast wave, all package variables in the path from P to *Root* have been reset. Then, the *P.started* remains FALSE until P creates its up-package. □

9. If *P.started* is TRUE, then there is at most one active package whose home process is P.

   Proof: *P.started* switches to TRUE only if $P.status = 2$. Then, all descendents and all ancestors has reset their package variables before P switches *P.status* to 2. So, there is no active package whose home is P hosted by these processes. Moreover, there is no such package anywhere else, otherwise in $O(n)$ rounds, some process R will satisfy $R.status = 0$ by Lemma 8; a contradiction to Claim 1. Hence, when P is enabled to switches *P.started* to TRUE, there is no active package whose home process is P.

   Then, P creates its own package only once (when switching *P.started* to TRUE), and once a package has been copied, previous copies become redundant. □

10. If $Q \in Chldrn(P)$ and if $P.up\_pkg \neq \perp$, then either $Q.up\_pkg.weight \geq P.up\_pkg.weight$ or $Q.up\_done$.

    Proof: Min-heap order is maintained, so that up-packages reach *Root* in weight order. □

11. Let p be the number of processes R such that *R.started* is FALSE or $R \notin \mathcal{S}$, and q be the number of active up-packages in $\mathcal{S}$. If $Root.status \in \{1, 2\}$, then $p + q + counter = n$, the size of $\mathcal{T}$.

Proof: At the start of each epoch, $p = n$ and $q = counter = 0$. Each time a process executes Action `CreateUpPkg`, p is decremented and q is incremented. Each time *Root* executes Action `CreateDownPkg`, q is decremented and *counter* is incremented. At the end (that is, the last configuration before *Root* takes status 3), $p = q = 0$ and $counter = n$. □

12. If $P.weight$ is the i^{th} smallest weight in $\mathcal{T}$, then $i > counter$ if and only if either $P.started$ is FALSE, or there is an active up-package whose home process is P.

    Proof: From Claims 7-11. □

13. If $P.started$ is TRUE and $P.rank$ is not the correct rank of P, then there is an active package in $\mathcal{S}$ whose home process is P.

    Proof: From the previous claim, the active up-package whose home is P will cause the creation of a down-package whose home is P with the correct rank value. □

14. If $P.status = 3$, then $P.started$ and $P.up\_done$ are TRUE, $P.up\_pkg$ and $P.down\_pkg$ are both redundant, and $P.rank$ has the correct value.

    If the status of P is 3, then P has completed its role in the epoch.

    Proof: *Root* is the first process to change its status to 3 during the epoch (by Action `BroadCast`), hence when P changes its status to 3, $Root.up\_done$ is TRUE, which, in turn, implies that $P.up\_done$ is TRUE (Claim 5). Moreover, if $P.up\_done$, then $P.started$ and $P.up\_pkg$ are redundant (see Action `EndUpPkg`). P gets status 3 only if there is no active down-package in the path from the root to P, so $P.down\_pkg$ is redundant, too (see Action `BroadCast`). Finally, by Claim 13, $P.rank$ has the correct value. □

Infinitely many complete epochs are executed, and during each of these epochs, all processes switch to status 3. By Claim 14, we thus have the following theorem:

Theorem 11. RANK *is self-stabilizing, computes the ranking of all processes in $O(n)$ rounds from an arbitrary initial configuration, and works under the weakly fair scheduler.*

References

[1] E. W. Dijkstra, Self-stabilizing systems in spite of distributed control, Commun. ACM 17 (11) (1974) 643–644.

[2] S. Dolev, Self-Stabilization, The MIT Press, 2000.

[3] P. Flocchini, A. M. Enriques, L. Pagli, G. Prencipe, N. Santoro, Point-of-failure shortest-path rerouting: Computing the optimal swap edges distributively, IEICE Transactions 89-D (2) (2006) 700–708.

[4] A. K. Datta, S. Gurumurthy, F. Petit, V. Villain, Self-stabilizing network orientation algorithms in arbitrary rooted networks, Stud. Inform. Univ. 1 (1) (2001) 1–22.

[5] P. Chaudhuri, H. Thompson, Self-stabilizing tree ranking, Int. J. Comput. Math. 82 (5) (2005) 529–539.

[6] B. Bourgon, A. K. Datta, V. Natarajan, A self-stabilizing ranking algorithm for tree structured networks, in: Proceedings of the First Workshop on Self-Stabilizing Systems (WSS'95), 1995, pp. 23–28.

[7] T. Herman, I. A. Pirwani, A composite stabilizing data structure, in: A. K. Datta, T. Herman (Eds.), Self-Stabilizing Systems, 5th International Workshop (WSS), Vol. 2194 of Lecture Notes in Computer Science, Springer, 2001, pp. 167–182.

[8] T. Herman, T. Masuzawa, A stabilizing search tree with availability properties, in: Fifth International Symposium on Autonomous Decentralized Systems (ISADS 2001), 2001, pp. 398–405.

[9] D. Bein, A. K. Datta, V. Villain, Snap-stabilizing optimal binary search tree, in: T. Herman, S. Tixeuil (Eds.), Self-Stabilizing Systems, 7th International Symposium (SSS), Vol. 3764 of Lecture Notes in Computer Science, Springer, 2005, pp. 1–17.

[10] S. Dolev, M. G. Gouda, M. Schneider, Memory requirements for silent stabilization, Acta Inf. 36 (6) (1999) 447–462.

[11] A. K. Datta, L. L. Larmore, S. Devismes, K. Heurtefeux, Y. Rivierre, Self-stabilizing small k-dominating sets, IJNC, International Journal of Networking and Computing 3 (1) (2013) 116–136.

[12] G. Tel, Introduction to distributed algorithms (2nd Ed.), Cambridge University Press, 2000.