L R I

ROUTE PRESERVING STABILIZATION

JOHNEN C / TIXEUIL S

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud-LRI

03/2003

Rapport de Recherche N° 1353

Route Preserving Stabilization

Colette Johnen   Sébastien Tixeuil

LRI-CNRS UMR 8623, Université Paris-Sud XI, 91405 Orsay cedex, France.

Abstract

A distributed system is *self-stabilizing* if it returns to a legitimate state in a finite number of steps regardless of the initial state, and the system remains in a legitimate state until another fault occurs. A routing algorithm is *loop-free* if, a path being constructed between two processors p and q, any edges cost change induces a modification of the routing tables in such a way that at any time, there always exists a path from p to q.

We present a self-stabilizing loop-free routing algorithm that is also *route preserving*. This last property means that, a tree being constructed, any message sent to the root is received in a bounded amount of time, even in the presence of continuous edge cost changes. Also, and unlike previous approaches, we do not require that a bound on the network diameter is known to the processors that perform the routing algorithm. We guarantee self-stabilization for many metrics (such as minimum distance, shortest path, best transmitter, depth first seach metrics, etc.), by reusing previous results on r-operators.

Key words: Distributed algorithms, fault-tolerance, self-stabilization, loop-free routing, route preserving routing

Résumé

Un système réparti est *auto-stabilisant* s'il retourne à un état légitime en un nombre fini d'étapes indépendamment de son état initial, et qu'il reste dans un état légitime jusqu'à ce qu'une nouvelle défaillance survienne. Un algorithme de routage est *sans boucles* si, un chemin étant construit entre deux processeurs p et q, tout changement de coût des arêtes induit une modification des tables de routage de telle manière qu'à chaque instant, il existe toujours un chemin de p à q.

Nous présentons un algorithme de routage sans boucles auto-stabilisant qui *préserve les routes*. Cette dernière propriété signifie que, un arbre de routage étant construit, tout message envoyé à la racine de l'arbre est reçu en temps fini, même en présente de changements continus des poids des arêtes. De plus, et contrairement aux approches précédentes, notre algorithme ne nécessite pas que les processeurs connaissent une borne sur le diamètre du système. Nous garantissons l'auto-stabilisation pour de multiples métriques (distance minimale, plus court chemin, meilleurs retransmetteurs, parcours en profondeur d'abord, etc.) en réutilisant des résultats antérieurs sur les r-opérateurs.

Mots clef: Algorithmes répartis, tolérance aux fautes, auto-stabilisation, routage sans boucles, routage préservant les routes

Contents

Chapter 1

Introduction

In 1974, Dijkstra pioneered the concept of self-stabilizatio in a distributed network [Dij74]. A distributed system is self-stabilizing if it returns to a *legitimate* state in a finite number of steps regardless of the initial state, and the system remains in a legitimate state until another fault occurs. Thus, a self-stabilizing algorithm tolerates transient processor faults. These transient faults include variable corruption, program counter corruption (which temporarily cause a processor to execute its code from any point), and communication channel corruption.

In the context of computer networks, resuming correct behavior after a fault occurs can be very costly [Per00]: the whole network may have to be shut down and globally reset in a good initial state. While this approach is feasible for small networks, it is far from practical in large networks such as the Internet. Self-stabilization provides a way to recover from faults without the cost and inconvenience of a generalized human intervention: after a fault is diagnosed, one simply has to remove, repair, or reinitialize the faulty components, and the system, by itself, will return to a good global state within a relatively short amount of time.

In the context of the communication networks, [VJ00] showed that crash and restart failures may lead the distributed systems into an arbitrary configurations, highlighting the need for algorithms that are resilient to those failures (such as self-stabilizing algorithms). Fault-tolerance is an important issue in the design of network routing protocols since the topology changes due to the link/node failures and repairs. Many existing routing protocols are self-stabilizing (for example [Per00] reports that RIP and OSPF have been proved self-stabilizing by Nancy Lynch).

Related works The primary task of a routing protocol is to generate and maintain routing table with the appropriate desirable property. This property is usually stated in terms of maximizing a particular *routing metric*. In [Dol97] is presented a self-stabilizing routing protocols on Breath-first path metric (shortest path metric where all edge weighs are 1). An optimal self-stabilizing algorithm for the shortest path metric is presented in [AKM+93]. In [GS99], a general self-stabilizing protocol that computes routing table for several routing metrics (for instance shortest path, maximal bandwidth path) on id-based networks is presented. In [DT01], another approach for routing is taken by means of r-operators (for minimum distance, shortest path, best transmitter, depth first search metrics). A common drawback of
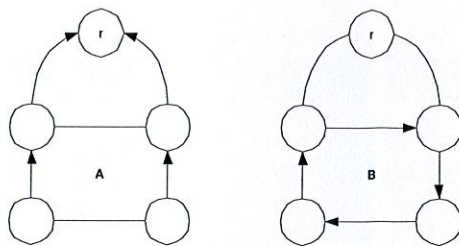
1

Figure 1.1: Example of self-stabilizing routing

these algorithms is that they are not *loop-free*: assume that a tree is initially present in the network, then if edge costs change, the self-stabilization property of the algorithm makes it return to a configuration where a tree is built. However, in the reconfiguration phase, it is possible that a cycle appears (see Figure 1.1). If a message is sent during this phase, it may loop in the network.

To circumvent this problem, self-stabilizing loop-free algorithms were developed. Assume that there exists a path between two nodes p and q. After any edges cost change, the loop-free protocol modifies its routing tables to compute the new optimal path. During this re-building phase, they always exists a path from p to q. In [AGH90] are presented two self-stabilizing routing protocols with shortest path metric. One of these protocols is loop-free, but requires unbounded memory per processor. In [CG02], a general self-stabilizing loop-free protocol is presented (using the formalism of [GS99]). The known self-stabilizing loop-free protocols (such as [AGH90, CG02]) suffer from two drawbacks: *(i)* they require that an upper bound on the diameter of the network is known to make them stabilizing, and *(ii)* if edge costs change often, it is possible that messages sent actually loop in the network. For example, in Figure 1.2, at each step there is a tree towards r that is constructed, yet message m does not reach r during configurations **A** to **F**. Since configurations **A** and **F** are the same, it is possible that m loops forever in the network if edge costs continue changing.

Our contribution We present a self-stabilizing loop-free routing algorithm that is also *route preserving*. Our solution is self-stabilizing so that it can recover from any transient failure once faults cease to appear; it is loop-free so that, a tree being constructed, a tree remains forever in the network even with continuous edge cost changes; it is route preserving so that, a tree being constructed, any message sent to the root is received within a bounded amount of time, even in the presence of continuous edge cost changes. This last feature is by providing a *routing policy* that permits to interleave consistently routing table updates and message forwarding actions. Figure 1.3 captures that our system recovers from transient faults (denoted by dotted lines in the figure) by eventually reaching a shortest path configuration provided that no new faults occur. In addition, edge cost changes (dashed lines in the figure) preserve the route preserving predicate in the sense that only route preserving configurations can be reached (denoted by grayed circles in the figure).

Also, and unlike [AGH90] and [CG02], we do not require that a bound on the network diameter is known to all processors in the network. This makes our solution more adaptive
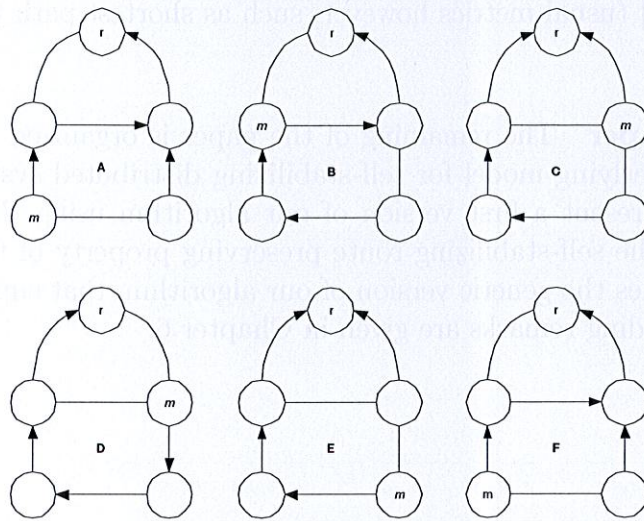
to network changes. Also in [CG02], we investigate self-stabilization for static routing trees as a solution to maintain shortest paths. This is achieved by depth first search traversal steps, whose principle approach is based on r-operators (see [DT00]). The set of metrics that are suggested is distinct from that of [GG03], namely, they do not have as much an abstract quality as they are in itself properties (approaches).

Outline of the presentation The rest of the chapters is organized as follows. In Chapter 2, we present the underpinned foundation for self-stabilizing distributed systems. For each of chapter. In Chapter 3, we develop a self-stabilizing algorithm with respect to the shortest path metric, and prove the self-stabilizing route preserving property of that algorithm in Chapter 4. Chapter 6 provides the main invention in our algorithm and suggests as using its use as co-operators. Concluding remarks are made in Chapter 6.

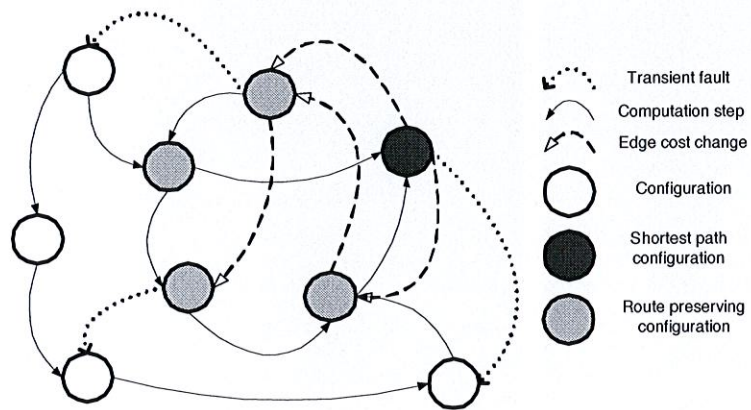Figure 1.2: Example of self-stabilizing loop-free routing

Figure 1.3: Properties of route preserving stabilization

3

to network changes. As in [CG02], we guarantee self-stabilization for many metrics (such as minimum distance, shortest path, best transmitter, depth first search metrics, etc.). Since our approach is based on r-operators (see [DT01]), the set of metrics that we support is distinct from that of [CG02] (usual metrics however, such as shortest path tree, are common to both apporaches).

Outline of the paper   The remaining of the paper is organized as follows. In Chapter 2, we present the underlying model for self-stabilizing distributed systems. For sake of clarity, in Chapter 3, we present a first version of our algorithm using the intuitive shortest path metric, and prove the self-stabilizing route preserving property of this algorithm in Chapter 4. Chapter 5 provides the generic version of our algorithm, that supports as many metrics as r-operators. Concluding remarks are given in Chapter 6.

Chapter 2

Model

Distributed systems A distributed system S is modeled as a collection of processors linked with communication media allowing them to exchange information. A distributed system is a connected graph $G = (V, E)$, where V is the set of nodes ($|V| = n$) and E is the set of links or edges. Each node $i \in V$ represents a processor, P_i. (For the sake of simplicity, we will use i and P_i interchangeably to represent the processor.) Each edge, denoted by a pair $(i, j) \in E$, represents a communication link between P_i and P_j, where P_i and P_j are called *neighbors*. Any ordered tuple of successive links $((i, k), (k, s), .., (l, j))$, represents a *path* $p_{i,j}$ between P_i and P_j. If two nodes P_i and P_j are connected by some path $p_{i,j}$ (note that $p_{i,i}$ is a path), P_j is said to be *reachable* from P_i.

The processors asynchronously execute their programs consisting of a set of variables and a finite set of rules. The variables are part of the shared register which is used to communicate with the neighbors. A processor can write only into its own shared register and can read only from the shared registers, owned by the neighboring processors or itself. So, variables of a processor can be accessed by the processor and its neighbors.

The *state* of a processor is defined by the values of its local variables. A *configuration* of a distributed system $G = (V, E)$ is an instance of the states of its processors. The set of configurations of G is denoted as C.

Processor actions change the global system configuration. Moreover, several processor actions may occur at the same time. A *computation* e of a system G is defined as a *weakly fair, maximal* sequence of configurations $c_1, c_2, \dots$ such that for $i = 1, 2, \dots$, the configuration c_{i+1} is reached from c_i by a single step of at least one processor. During a computation step, one or more processors execute a step and a processor may take at most one step. *Weak fairness* of the sequence means that if any action in G is continuously enabled along the sequence, it is eventually chosen for execution. *Maximality* means that the sequence is either infinite, or it is finite and no action of G is enabled in the final global state.

Let C be the set of possible configurations and $\mathcal{E}$ be the set of all possible computations of a system G. Then the set of computations of G starting with a particular *initial configuration* $c_1 \in C$ will be denoted by $\mathcal{E}_{c_1}$. Every computation $e \in \mathcal{E}_{c_1}$ is of the form $c_1, c_2, \dots$ The set of computations of $\mathcal{P}$ whose initial configurations are all elements of $B \subseteq C$ is denoted as $\mathcal{E}_B$.

Thus, $\mathcal{E} = \mathcal{E}_\mathcal{C}$.

We introduce the concept of an *attractor* to define self-stabilization. Intuitively, an attractor is a set of configurations of a system G that "attracts" another set of configurations of G for any computation of G.

Definition 1 (Attractor) *Let B_1 and B_2 be subsets of $\mathcal{C}$. Then B_1 is an attractor for B_2 if and only if $\forall e \in \mathcal{E}_{B_2}$, $(e = c_1, c_2, \ldots)$, $\exists i \geq 1$: $c_i \in B_1$.*

The shortest path maintenance is a static problem. The set of configurations that matches the specification of static problems is called the set of *legitimate* configurations, denoted as $\mathcal{L}$. The remainder $\mathcal{C} \setminus \mathcal{L}$ denotes the set of *illegitimate* configurations.

Definition 2 (Self-stabilization) *A distributed system $\mathcal{S}$ is called self-stabilizing if and only if there exists a non-empty set $\mathcal{L} \subseteq \mathcal{C}$ such that the following conditions hold: (i)$\forall e \in \mathcal{E}_\mathcal{L}$, $(e = c_1, c_2, \ldots)$, $\forall i \geq 1$, $c_i \in \mathcal{L}$ (closure). (ii) $\mathcal{L}$ is an attractor for $\mathcal{C}$ (convergence).*

The program for a protocol consists of a sequence of rules: $< rule > \cdots < rule >$. Each *rule* is the form: $< guard > \longrightarrow < action >$. A *guard* is a boolean expression over the variables of a node and its neighborhood. An *action* is allowed to update the variables of the node only. Any rule whose guard is *true* is said to be *enabled*. A node with one or more enabled rule is said to be *privileged* and may make a *move* executing the action corresponding to the chosen enabled rule. We do not make any assumption on the policy adopted by a node when more than one rule guards are satisfied.

Chapter 3

The route preserving self-stabilizing shortest path algorithm

3.1 Problem to be solved

In this section, we present the problem to be solved by our algorithm. The purpose of our contribution is threefold: we wish to build a shortest path tree rooted at r (the shortest path problem), we wish to be able to recover from any transient fault (the self-stabilization requirement), and we want that when routes are constructed, messages can be routed during changes of edge costs (the route reserving requirement).

The shortest path problem We associate to each link (i, j) of $G = (V, E)$ a positive integer $c_{i,j}$ that denotes the cost of a communication through this link in both direction (*i.e.* from i to j and from j to i). Each processor i has the following inputs: $\mathcal{N}_i$ is the locally ordered set of neighbors of i, and $c_{i,j}$ is the cost of edge between processors i and j.

On each processor i, the following variables are available: p_i is a pointer to a particular neighbor of i, that is equal to the next processor $j \in \mathcal{N}_i$ on the current path from i to r, and w_i is an integer variable that is equal to the cost of the current path from i to r.

Given a source r, the shortest path problem consists in finding for each processor i a path from i to r (or from r to i) whose overall cost is minimal. Each path is induced by the p variables at all nodes $i \neq r$. We denote by $\dot{w}_i$ the cost of the shortest path from i to r (note that $\dot{w}_r = 0$). In a shortest path from i to r, $\dot{w}_i$ is equal to the minimum of the costs at a neighbor j of i plus the cost from j to i for any possible neighbor j of i. Formally, $\forall i \neq r$, $\dot{w}_i = Min_{j \in \mathcal{N}_i}(\dot{w}_j + c_{i,j})$. We now define a shortest path configuration (that must be eventually reached by any algorithm that wishes to solve the shortest path problem):

Definition 3 (Shortest-path Configuration) *Let $\mathcal{SP}$ be the following predicate on system configurations: $\mathcal{SP} \equiv \{\forall i \in V, w_i = \dot{w}_i\}$. A shortest path configuration is a configuration that satisfies $\mathcal{SP}$.*

7

The self-stabilization requirement Not only our algorithm eventually provides a shortest path configuration, it also ensures that the initial state can be arbitrary. Once the system is stabilized, the set of p pointers should induce a shortest path tree rooted at r.

Due to the self-stabilization property of our algorithm, it computes for each processor $i \neq r$ a shortest path to r from any initial configuration configuration under any weakly fair scheduler. Our algorithm does not make any assumption about the initial value of w_i or p_i on any processor i. We now define the legitimate configurations for our purpose using a *legitimate predicate* $\mathcal{LP}$.

Definition 4 (Legitimate Configuration) *Let* $\mathcal{LP}$ *be the following predicate on system configurations:* $\mathcal{LP} \equiv \{(\forall i \neq r, (w_i = \dot{w}_i) \wedge (w_i = w_{p_i} + c_{i,p_i})) \wedge (w_r = 0)\}$*. A legitimate configuration is a configuration that satisfies* $\mathcal{LP}$*.*

The route preserving requirement We wish to preserve a route towards the root r once a path is constructed, even if edge costs are modified. For that purpose, we define so that the stabilizing algorithm wishes to change node variables.

Definition 5 (Task preservation under input change) *Let* TS *be a task specification, let* TC *be a set of changes of inputs in the system. A distributed system* $\mathcal{S}$ *preserves* TS *under* TC *if and only if there exists a non-empty set of configurations* $C \subseteq \mathcal{C}$ *such that the following conditions hold: (i)* C *is closed, (ii)* C *is closed under any change in* TC*, and (iii) every computation starting from* $c \in C$ *eventually satisfies* TS*.*

For our purpose, the task to be solved (TS) is to be able to route a message for any node i to the root r, while the set of changes of inputs (TC) is the set of possible variation of edge costs.

3.2   Description

Each processor i performs several kinds of actions with different objectives:

Route finding i whishes to reach a configuration where every node maintains a route towards r (so that messages can be sent to r). This goal is achieved by having the weight of each node (the w_i variable) set to a value that is strictly greater than that of its parent (denoted by the w_{p_i} variable).

Route preserving The rules of the algorithms that are executed *after* a route is present towards r from every node i have to preserve a route towards r from every node i. Also, arbitrary changes of edge costs must not break the such defined routes (for the route preserving requirement).

Shortest path finding i wishes to reach a shortest path configuration.

Self-stabilization i wishes to reach a legitimate configuration (for the self-stabilization requirement). For that purpose, if $i = r$, then i updates w_i so that $w_i = 0$, and if $i \neq r$, i updates p_i and w_i so that $w_i = Min_{j \in \mathcal{N}_i}(w_j + c_{i,j})$.

8

Formal description

Each processor i (except the root) has the following variables: rw_i (the current value broadcast in i's sub-tree) is an integer, and p_i (the parent of i) is a pointer to a neighbor of i. Each processor i (including the root) has the following variables: w_i (the weight of i) is an integer, and st_i (the status of i) takes value in $\{P, N\}$.

All processors has the following local macro: $\hat{w}_i$ is the best possible value of w_i according to the current configuration (see table below). Let j be a child of i ($p_j = i$). j is called a *descendant* of i iff $w_j > w_i$. Each processor (except the root) has the following local macros: $\mathcal{D}_i$ is the set of the descendants of i in the tree, $\hat{p}_i$ is the best possible parent i according to the current configuration if a such neighbor exists otherwise the value of $\hat{p}$ is $\bot$ (The best parent j should have the neutral status and should have a weight that minimizes the new weight value of i: $\hat{w}_j + c_{i,j}$), and ubw_i is the upper bound of w_i (that i might take and still preserve convergence towards a legitimate state) in the current configuration (see table below).

$$\hat{w}_i \quad \equiv \quad \begin{cases} \min_{j \in \mathcal{N}_i}(w_j + c_{i,j}) & \text{if } i \neq r \\ 0 & \text{otherwise} \end{cases}$$

$$\hat{p}_i \quad \equiv \quad \begin{cases} \min_{j \in \mathcal{N}_i}(j :: (w_j = \hat{w}_i - c_{i,j}) \wedge (st_j = N)) & \text{if there exists } j \in \mathcal{N}_i :: \\ & \quad (w_j = \hat{w}_i - c_{i,j}) \wedge (st_j = N) \\ \bot & \text{otherwise} \end{cases}$$

$$\mathcal{D}_i \quad \equiv \quad \{j \in \mathcal{N}_i :: p_j = i \wedge w_j > w_i\}$$

$$ubw_i \quad \equiv \quad \begin{cases} \min_{j \in \mathcal{D}_i}(w_j - c_{i,j}) & \text{if } \mathcal{D}_i \neq \emptyset \\ \infty & \text{otherwise} \end{cases}$$

Each processor (except the root) has the following local predicates: End_PIF is used to avoid that a processor increases its weight (i.e. to perform R_3 action) simultaneously with one of its descendants, $Safe\_MOVE$ is used to avoid that a processor i takes the status P (i.e. to perform R_2 action) when it could change its parent, and $Safe\_INC$ is verified by i when it needs to increase its weight (see table below).

$$End\_PIF_i \quad \equiv \quad (\forall j \in \mathcal{D}_i, st_j = N)$$

$$Safe\_MOVE_i \quad \equiv \quad \begin{pmatrix} & ((\hat{w}_i < w_i) \vee ((w_i = \hat{w}_i) \wedge (\hat{p}_i \neq p_i))) \\ \wedge & (\hat{p}_i \neq \bot) \end{pmatrix}$$

$$Safe\_INC_i \quad \equiv \quad \begin{pmatrix} & ((w_i < rw_{p_i} + c_{p_i,i}) \wedge (st_{p_i} = P)) \\ \vee & (w_i < w_{p_i} + c_{p_i,i}) \end{pmatrix}$$

The rules of our algorithm are the following:

1. On the root r processor:

9

- $R_0 :: (w_r \neq 0) \vee (st_p \neq \mathsf{N}) \rightarrow$
  $w_r \leftarrow 0; st_r \leftarrow \mathsf{N}$

2. On any other processor i:

- $R_1 :: (st_i = \mathsf{N}) \wedge Safe\_MOVE_i \rightarrow$
  $w_i \leftarrow \hat{w}_i; \; rw_p \leftarrow w_i; \; p_i \leftarrow \hat{p}_i$
- $R_2 :: (st_i = \mathsf{N}) \wedge \neg Safe\_MOVE_i \wedge Safe\_INC_i \rightarrow$
  $rw_i \leftarrow rw_{p_i} + c_{p_i,i}; \; st_i \leftarrow \mathsf{P}$
- $R_3 :: (st_i = \mathsf{P}) \wedge End\_PIF_i \wedge (ubw_i \geq rw_i) \rightarrow$
  $w_i \leftarrow rw_i; \; st \leftarrow \mathsf{N}$
- $R_4 :: (rw_i < w_i) \rightarrow rw_i \leftarrow w_i$

The R_0 and R_4 rules are designed to ensure that eventually the variable values on a processor are locally correct. The R_1 action allows a processor to get the best value for its w and p variables according to the state of its neighbor. The R_2 action is executed when i needs to increase its weight, while the actual increase is achieved by executing the R_3 action.

Observation 1 *The weight of a processor increases only after that this processor has performed the rule R_3. Actions R_1, R_2 and R_3 are mutually exclusive: a processor cannot simultaneously verify two rule guards of R_1, R_2 or R_3. Rule R_0 is performed at most once. After any action on i, we have $w_i \leq rw_i$ therefore the action R_4 is performed at most once by any processor.*

Lemma 1 *When a processor i performs R_3, none of its descendants and parent can perform R_3 simultaneously with i.*

Proof: When the R_3 guard holds, i's descendants and parent have Status N. They cannot satisfy the R_3 guard. □

Chapter 4

Proof of correctness

In this section, we show that our algorithm is self-stabilizing for the shortest path tree problem. We also prove that there exists a *route preserving* set of configurations $\mathcal{RP}$ that ensure routing of messages towards the root r and that is preserved for any change of edge cost.

4.1 Route preserving proof

We wish to find a non-empty set of configurations $\mathcal{RP}$ that verifies the following properties: *(i)* $\mathcal{RP}$ is not empty, *(ii)* $\mathcal{RP}$ is closed (any action of any processor executing our algorithm preserves $\mathcal{RP}$), *(iii)* $\mathcal{RP}$ is closed under any edge cost changes, and *(iv)* every computation starting from a configuration in $\mathcal{RP}$ is guaranteed to deliver the message at r after finite time. Once $\mathcal{RP}$ is verified, any message sent to r reaches it. Therefore, once $\mathcal{RP}$ is verified, our algorithm guarantees that *(i)* any message towards r reaches its destination, even when the edge costs vary, and *(ii)* routes to r are eventually the costless ones. We define $\mathcal{RP}$ as the set of configurations where Predicates $\text{Pr}_i^{\mathcal{RP}1}$ and $\text{Pr}_i^{\mathcal{RP}2}$ hold at each $i \in V$. Those predicates are defined as follows:

Definition 6 *Let $\text{Pr}_i^{\mathcal{RP}1}$ be the following predicate on processor state:*

$$\underset{i}{\text{Pr}}^{\mathcal{RP}1} \equiv \begin{cases} rw_i \geq w_i & \text{if } i \neq r \\ w_r = 0 \wedge st_r = \mathsf{N} & \text{otherwise} \end{cases}$$

Definition 7 *Let $\text{Pr}_i^{\mathcal{RP}2}$ be the following predicate on processor state:*

$$\underset{i}{\text{Pr}}^{\mathcal{RP}2} \equiv \begin{cases} w_{p_i} < w_i & \text{if } i \neq r \\ w_r = 0 & \text{otherwise} \end{cases}$$

Our algorithm ensures that increasing a weight is safe: *(i)* i cannot gain a new child while performing its weight increase action, *(ii)* all descendants j of i still satisfy Predicate $\text{Pr}_j^{\mathcal{RP}2}$ after any action of i, and *(iii)* i changing its p_i variable also preserves Predicate $\text{Pr}_i^{\mathcal{RP}2}$. In our algorithm, the st_i variable at processor i helps to guarantee Properties *(i)* and *(iii)*. The rw_i variable of i represents the maximal value that i can take and its descendants still satisfy

11

Predicate $\mathrm{Pr}_i^{\mathcal{RP}_2}$ predicate. We now prove that the preserving predicate $\mathrm{Pr}_i^{\mathcal{RP}_1} \wedge \mathrm{Pr}_i^{\mathcal{RP}_2}$ is closed.

Lemma 2 $A^{\mathcal{RP}} \equiv \{\forall i \in V, \mathrm{Pr}_i^{\mathcal{RP}_1} \text{ holds}\}$ *is an attractor for* true.

Proof: A processor i that does not verify $\mathrm{Pr}_i^{\mathcal{RP}_1}$ verifies forever R_0 or R_4 guard. By the weak fairness hypothesis, every enabled processor i executes an action. After this action, we have $rw_i \geq w_i$ (observation 1). $\square$

Observation 2 *In* $A^{\mathcal{RP}}$, *no processor performs the actions* R_0 *and* R_4.

Lemma 3 *In* $A^{\mathcal{RP}}$, *on any processor* i, $\mathrm{Pr}_i^{\mathcal{RP}_2}$ *is closed.*

Proof: Predicate $\mathrm{Pr}_i^{\mathcal{RP}_2}$ may becomes false its value after that either: *(i)* i executes R_1 or, *(ii)* p_i executes R_3.

Just before i executes R_1 to choose j as its parent, j has status N. Therefore j cannot increase it weight (i.e. perform action R_3) simultaneously with i's action. Thus, we have $w_i \geq w_j + c_{i,j}$, after i's action. Let us now study the predicate value on i when its parent j performs the action R_3. When j performs the action R_3, Its descendants (i.e. children of j that verify the $\mathrm{Pr}^{\mathcal{RP}_2}$ predicate) cannot perform the action R_3 (lemma 1). A descendant of j still verifies $\mathrm{Pr}^{\mathcal{RP}_2}$ after the action of j ($w_j < w_i$ where $i \in \mathcal{D}_j$). During the j action, j cannot gain a new descendant via action R_1 because j has the status P. $\square$

Lemma 4 $\mathcal{RP}$ *is not empty and is closed.*

Any configuration of $\mathcal{LP}$ (the non-empty set of legitimate configurations) is in $\mathcal{RP}$, so $\mathcal{RP}$ is not empty. By lemmas 2 and 3, $\mathcal{RP}$ is closed.

Lemma 5 $\mathcal{RP}$ *is closed under any edge cost change.*

Proof: Predicate $\mathrm{Pr}^{\mathcal{RP}_1}$ only involves local variables of a processor i (it does not involve the local variables of i's neighbors and edge cost values). Predicate $\mathrm{Pr}^{\mathcal{RP}_2}$ only involves local variables of a processor i and the local variables of i's neighbors. Therefore, the $\mathcal{RP}$ set of configuration is independent of edge cost values. $\square$

There remains to show that every computation starting from a configuration in $\mathcal{RP}$ is guaranteed to deliver the message at r after finite time. For that purpose, we consider the following *routing policy* (denoted by RP) for every message: *(i)* a message to r from a processor $i \neq r$ is sent to p_i, *(ii)* on a processor i, R_3 is performed only when i does not have any message to send to r.

Lemma 6 *There exists a routing policy* RP *such that once a configuration in* $\mathcal{RP}$ *is reached, there exists a bound on the number of hops number for any message sent towards* r.

Proof: The current weight of a message m on a processor i is the weight of i. According to the definition of $\mathcal{RP}$ and RP, the two following assertions hold: *(i)* the weight of a message decreases each time the message is transmitted to the next processor on the path, and *(ii)* the weight of a message m is 0 if and only if m has reached r.

Indeed, the weight of m on processor i can only increase when i performs action R_3 (R_3 is the only action that may increase the weight of a processor). The routing policy RP guarantees that a processor does not have any waiting message to r when it performs R_3. The weight of message never increases, therefore, the weight of a message is an upper bound on the number of processors that are traversed before reaching r. $\square$

We now consider a new routing policy RP' that ensures fairness of rules execution and message forwarding. Point *(ii)* of routing policy RP is changed as follows. A processor i holding the R_3 guard executes the following actions *(i)* i sends a message to its descendants and to its application layer to have them keep any message to r; *(ii)* i sends to p_i any waiting message to r in its buffer and its incoming channels from its descendants, *(iii)* i performs R_3, and *(iv)* i authorizes its children to continue routing of their own user messages.

With the routing policy RP', a processor i that prevents the R_3 rule from executing is eventually message free (and can then execute R_3). As expected, configurations in $\mathcal{RP}$ are secured under any series of modifications of edge costs. Moreover, any computation starting from a configuration in $\mathcal{RP}$ guarantees that any message sent to r reaches r in a bounded number of hops (this bound is the weight of the sender processor at the sending time).

4.2 Self-stabilization proof

From now onwards, we consider that the initial configuration is a configuration in $A^{\mathcal{RP}}$. Such a configuration is eventually reached by Lemma 2. Also, we assume that edge costs remain constant. This does not break the route preserving requirement, since when $A^{\mathcal{RP}}$ hold, routing can always be performed. However, self-stabilization towards a shortest path tree is only guaranteed if edge costs remain constant for a sufficiently long period of time.

The overview of our proof is as follows. First, we show that starting from a configuration in $A^{\mathcal{RP}}$, the weight of each processor eventually gets lower bounded by the weight of its parent plus the cost of the link. Then, we show that whenever a processors initiates a propagation of information to its subtree, this propagation eventually gets back to it (a processor with Status P eventually gets Status N). Then, we prove that the weight of each processor eventually gets greater or equal to the optimal weight of this processor (when the shortest path tree is constructed). Finally, we prove that the algorithm converges to a configuration where a shortest path tree is built, and where local consistency also holds, which makes the configuration legitimate.

4.2.1 Local consistency

We define the local consistency predicate $\mathrm{Pr}_i^{\mathcal{LC}}$ as a refinement of Predicate $\mathrm{Pr}_i^{\mathcal{RP}_1} \wedge \mathrm{Pr}_i^{\mathcal{RP}_2}$.

Definition 8 *Let* $\mathrm{Pr}_i^{\mathcal{LC}}$ *be the following predicate on processor state:*

$$\mathrm{Pr}_i^{\mathcal{LC}} \equiv \begin{cases} \mathrm{Pr}_i^{\mathcal{RP}_1} \wedge (w_{p_i} + c_{i,p_i} \leq w_i) & \text{if } i \neq r \\ \mathrm{Pr}_r^{\mathcal{RP}_1} & \text{otherwise} \end{cases}$$

The proof of the following lemma is similar to that of Lemma 3.

Lemma 7 *In* $A^{\mathcal{RP}}$, *on any processor* i, *the predicate* $\mathrm{Pr}_i^{\mathcal{LC}}$ *is closed.*

Proof: Predicate $\mathrm{Pr}_i^{\mathcal{LC}}$ may becomes false when either: *(i)* i executes R_1 or, *(ii)* p_i executes R_3.

After i performs R_1, $\mathrm{Pr}_i^{\mathcal{LC}}$ is satisfied (see Action R_1). Now assume i satisfies $\mathrm{Pr}_i^{\mathcal{LC}}$. Let us study the predicate value on i after its parent performs Action R_3. Before Action R_3 of p_i is executed, we have $rw_{p_i} \leq ubw_i \leq w_i - c_{p_i,i}$. Then i cannot increase its weight (*i.e.* perform Action R_3) simultaneously with p_i's action. After Action R_3 of p_i is executed, we have $w_{p_i} \leq w_i - c_{p_i,i}$. $\qquad\square$

4.2.2 Liveness

In this section, we prove that any processor that gets Status P (at any stage of the algorithm execution from a configuration in $A^{\mathcal{RP}}$) eventually performs action R_3 and gets neutral status (i.e. N).

Definition 9 (Broadcast tree) *The* broadcast tree *of* i, *denoted by* BT_i, *is the directed subgraph* (V_i, E_i) *of* (V, E) *such that* $(V_i, E_i) \equiv (V_{i,n}, E_{i,n})$, *where* $(V_{i,n}, E_{i,n})$ *is recursively defined as follows:*

- $i \in V_{i,0}$,

- $\forall m \geq 1, j \in V_{i,m-1} \Rightarrow j \in V_{i,m}$,

- $\forall m \geq 1, (j \in V_{i,m-1}) \wedge (k \in \mathcal{D}_j) \wedge (st_k = \mathsf{P}) \Rightarrow k \in V_{i,m}$,

- $\forall m \geq 1, (j \in V_{i,m-1}) \wedge (j \in \mathcal{D}_{p_j}) \wedge (st_{p_j} = \mathsf{P}) \Rightarrow p_j \in V_{i,m}$,

- $\forall m \geq 1, (k \in V_{i,m}) \wedge (p_k \in V_{i,m}) \Rightarrow (p_k, k) \in E_{i,m}$.

Lemma 8 BT_i *is a directed tree.*

Proof: Let j be a processor of BT_i with $j \neq i$. By construction, the following properties are satisfied: *(i)* there exists a path in BT_i from j to i or from i to j (BT_i is a connected directed graph), *(ii)* by definition of E_i and $\mathcal{D}_k$, $(k, j) \in E_i \Rightarrow (p_j = k) \wedge (w_k < w_j)$ (BT_i is without cycle), and *(iii)* each processor of V_i has at most one in-going arc. Therefore BT_i is a tree. $\square$

Lemma 9 *Let* $s \in \mathcal{E}_{A^{\mathcal{RP}}}$. *Let* $i \in V$. *If* $st_i = \mathsf{P}$ *then* i *eventually performs* R_3 *in* s.

14

Proof: Let l be the root of BT_i. Assume that i remains forever in BT_i. Then, every processor $j \in E_i$ on the path from l to i is disabled (j has status P and End_PIF_j never hold). Thus l remains forever in BT_i. Let G_i be the subtree of BT_i that contains all processors that remain forever in BT_i. G_i is not empty, since it contains l. Let j be a leaf of G_i. We now consider the following predicates on processor states:

$$\Pr_k^\alpha \equiv (w_k \geq rw_{p_k} + c_{p_k,k})$$

$$\Pr_k^\beta \equiv (st_k = \mathsf{N} \wedge \Pr_k^\alpha)$$

A descendant k of j that does verify $\Pr_k^\beta$ still satisfy this predicate after executing any action (it may only perform R_1). Now, if $\forall k \in \mathcal{D}_j$, k satisfies $\Pr_k^\beta$, then j remains enabled forever (Action R_3). Eventually, j performs R_3 and quits G_i. Assume that j has a descendant k ($k \in \mathcal{D}_j$) that never satisfies $\Pr_k^\beta$. If $st_k = \mathsf{P}$, then k eventually performs R_3 (k does not remain forever in G_i). Therefore k never satisfies $\Pr_k^\alpha$. Then, either $st_k = \mathsf{N}$ or $st_k = \mathsf{P}$. If $st_k = \mathsf{P}$, then k eventually performs R_3. If $st_k = \mathsf{N}$, then k is able to perform R_2 or R_1 forever. If k performs R_1, then k is no more a descendant of j. Otherwise k would eventually perform R_2 followed by R_3. After this sequence, we have $w_k \geq rw_j + c_{j,k}$. Then, k satisfies $\Pr_k^\alpha$. Thus, a contradiction appears. $\square$

4.2.3 Lower bound on weight

Starting from any configuration in $A^{\mathcal{RP}}$, any computation of our algorithm eventually guarantees that the weight w_i of every processor i is greater or equal to the minimal cost of a path from i to r. Thus, each node i satisfies a lower bound predicate $\Pr_i^{\mathcal{LB}}$.

Definition 10 *Let $\Pr_i^{\mathcal{LB}}$ be the following predicate on processor state:*

$$\Pr_i^{\mathcal{LB}} \equiv (w_i \geq \dot{w}_i)$$

Lemma 10 $A^{\mathcal{LB}} \equiv \{\forall i \in V, \Pr_i^{\mathcal{LB}} \vee \Pr_i^{\mathcal{LC}} \text{ holds}\}$ *is an attractor for* $A^{\mathcal{RP}}$.

Proof: Let s be a computation of $\mathcal{E}_{A^{\mathcal{RP}}}$ where i never satisfies $\Pr_i^{\mathcal{LC}}$. If $st_i = \mathsf{P}$ then i eventually performs R_3 (by Lemma 9) and then $st_i = \mathsf{N}$. If $st_i = \mathsf{N}$ then $Safe\_INC_i$ holds. Then, i remains enabled until it performs an action (R_1 or R_2). If i performs R_1, then after the computation step, i verifies $\Pr_i^{\mathcal{LC}}$. $\Pr_i^{\mathcal{LC}}$ is a closed predicate (see Lemma 7).
Along s, if i never performs R_1, then i performs R_2 followed by R_3 infinitely often (by Lemma 9). Thus w_i is strictly increasing. Therefore, eventually w_i is greater than $\dot{w}_i$ and $\Pr_i^{\mathcal{LB}}$ holds onwards. $\square$

Lemma 11 *In* $A^{\mathcal{LB}}$, *for every processor* i, $\Pr_i^{\mathcal{LB}}$ *holds.*

Proof: Assume that there exists a processor i that do not satisfy $\mathrm{Pr}_i^{\mathcal{LB}}$. Obviously, we have $i \neq r$. In $A^{\mathcal{LB}}$, i satisfies $\mathrm{Pr}_i^{\mathcal{LC}}$ (i.e. $w_i \geq w_{p_i} + c_{p_i, i}$). We thus have $\dot{w}_{p_i} + c_{p_i, i} \geq \dot{w}_i$. Now, if p_i does satisfy $\mathrm{Pr}_{p_i}^{\mathcal{LB}}$ (i.e. $w_{p_i} \geq \dot{w}_{p_i}$) then i satisfies $\mathrm{Pr}_i^{\mathcal{LB}}$ (i.e. $w_i \geq \dot{w}_i$). We can conclude that $p_i \neq r$ and that p_i does not verify $\mathrm{Pr}_i^{\mathcal{LB}}$, and we have $w_i > w_{p_i}$.

This doing, we construct an infinite sequence of processors $i_1, i_2, i_3, \ldots$ such that $\forall n, p_n = i_{n+1}$ and $w_{i_n} > w_{i_{n+1}}$. As $\forall n, w_{i_n} > w_{i_{n+1}}$, all processors of the sequence are pairwise distinct. Since the total number of processors in the network is finite, the initial hypothesis is contradicted. $\qquad\square$

4.2.4 Towards legitimate configurations

In this section, we finally prove that starting from an arbitrary initial configuration, any computation reaches a legitimate and terminal configuration. This proves the self-stabilization property of our algorithm. We first define the shortest path local predicate $\mathrm{Pr}_i^{\mathcal{SP}}$ for every node i.

Definition 11 *Let $\mathrm{Pr}_i^{\mathcal{SP}}$ be the following predicate on processor states:*

$$\mathrm{Pr}_i^{\mathcal{SP}} \equiv (w_i = \dot{w}_i) \wedge (st_i = \mathsf{N})$$

Definition 12 $A^{\mathcal{SP}} \equiv \{\forall i \in V,\ \mathrm{Pr}_i^{\mathcal{SP}}\ holds\}$.

We wish to prove that $A^{\mathcal{SP}}$ is an attractor for $A^{\mathcal{LB}}$. The proof is by induction on the length of the minimal cost path of a processor to the root. We thus define the following set $\mathcal{X}_d$.

Definition 13 *Let d be a positive integer. Let $\mathcal{X}_d$ be the set of processors having a minimal cost path to the root of length d (in number of hops)*

For sake of the induction proof, we also weaken $A^{\mathcal{SP}}$ as a family of parametrized predicates $A_d^{\mathcal{SP}}$. Note that $A_n^{\mathcal{SP}} = A^{\mathcal{SP}}$, where n is the number of processors in the system.

Definition 14 *Let $A_d^{\mathcal{SP}} \equiv \{\forall i \in V,\ the\ predicate\ (i \in \mathcal{X}_d \Rightarrow \mathrm{Pr}_i^{\mathcal{SP}})\ holds\}$.*

Since $\mathcal{X}_0 = \{r\}$, $A_0^{\mathcal{SP}} \equiv \{\forall i,\ the\ predicate\ (i \in \mathcal{X}_0 \Rightarrow \mathrm{Pr}_i^{\mathcal{SP}})\ holds\}$ is an attractor for $A^{\mathcal{LB}}$. Note that $A_0^{\mathcal{SP}}$ is included in $A^{\mathcal{RP}}$.

Lemma 12 *Let d be an integer ($d \geq 0$) such that $A_d^{\mathcal{SP}}$ is an attractor for $A^{\mathcal{LB}}$. Then $A_{d+1}^{\mathcal{SP}}$ is an attractor for $A^{\mathcal{LB}}$.*

Proof: Let i be a processor of $\mathcal{X}_{d+1}$. Let s be a computation whose initial configuration is in $A_d^{\mathcal{SP}}$. i has a neighbor k in X_d such that $\dot{w}_i = \dot{w}_k + c_{k,i} = w_k + c_{k,i}$. According to Lemma 11, we have:

1. $\forall j \in \mathcal{N}_i, w_j \geq \dot{w}_j$, therefore $\hat{w}_i \geq \dot{w}_i$,

16

2. $\hat{w}_i \leq c_{k,i} + w_k = \dot{w}_i$, and

3. $w_i \geq \dot{w}_i$.

We conclude that $w_i \geq \hat{w}_i = \dot{w}_i$. If $st_i = \mathsf{P}$, i eventually performs R_3 (by Lemma 9).

Now $st_i = \mathsf{N}$. If $st_{p_i} = \mathsf{P}$ or if $w_i < w_{p_i} + c_{p_i,i}$, then $Safe\_MOVE_i$ holds (either $w_i > \hat{w}_i$ or i verifies $(w_i = \hat{w}_i) \wedge (p_i \neq \hat{p}_i)$). Thus if i verifies $Safe\_INC_i$ then i also verifies the predicate $Safe\_MOVE_i$. Then, i may only execute R_1.

Once $st_i = \mathsf{N}$, while $\Pr_i^{\mathcal{SP}}$ does not hold, i verifies the R_1 guard. By the weak fairness hypothesis, i eventually performs R_1. After this action, $\Pr_i^{\mathcal{SP}}$ holds.

Once $\Pr_i^{\mathcal{SP}}$ holds, i may only perform R_1. $\qquad\square$

According to the preceding proof, along any computation beginning in a configuration where $A_{d+1}^{\mathcal{SP}}$ holds, a processor of $\mathcal{X}_d$ can only perform R_1.

Corollary 1 $A^{\mathcal{SP}} \equiv \{\forall i \in V, \Pr_i^{\mathcal{SP}} \text{ holds}\}$ *is an attractor for* $A^{\mathcal{LB}}$.

We now define a local predicate $\Pr_i^{\mathcal{LC}}$ for node i belonging to a legitimate configuration.

Definition 15 *Let* $\Pr_i^{\mathcal{LP}}$ *be the following predicate on processor states:*

$$\Pr_i^{\mathcal{LP}} \equiv \Pr_i^{\mathcal{SP}} \wedge (w_i = w_{p_i} + c_{p_i,i})$$

Lemma 13 $A^{\mathcal{LP}} \equiv \{\forall i \in V, \Pr_i^{\mathcal{LP}} \text{ holds}\}$ *is an attractor for* $A^{\mathcal{SP}}$.

Proof: In $A^{\mathcal{SP}}$, the weight of every processor is the right one and all processors have the N status. Let i be a processor different from r. If i does not satisfy $\Pr_i^{\mathcal{LP}}$, then $w_i \neq w_{p_i} + c_{p_i,i}$. In $A^{\mathcal{SP}}$, i has a neighbor k such that $w_i = \dot{w}_i = \dot{w}_k + c_{k,i} = w_k + c_{k,i}$. Therefore, i can perform R_1 forever and choose k as a parent. After this action, i satisfies $\Pr_i^{\mathcal{LP}}$.
In $A^{\mathcal{SP}}$, processors do not change their weight value, thus $\Pr^{\mathcal{LP}}$ is a closed predicate. $\qquad\square$
In $A^{\mathcal{LP}}$, $\mathcal{LP}$ is satisfied. Moreover, all configurations of $A^{\mathcal{LP}}$ are terminal configurations.

Theorem 1 *Any computation* $s \in \mathcal{E}$ *is finite. The terminal configuration of* s *satisfies* $\mathcal{LP}$.

Chapter 5

The generic algorithm

r-operators   Following work of Tel concerning wave algorithms (see [Tel94]), an *infimum* $\oplus$ (hereby called an *s-operator*) over a set $\mathbb{S}$ is an associative, commutative and idempotent (*i.e.* $x \oplus x = x$) binary operator. Such an operator defines a partial order relation $\leq_\oplus$ over the set $\mathbb{S}$ by: $x \leq_\oplus y$ if and only if $x \oplus y = x$. We denote by $e_\oplus$ a greatest element on $\mathbb{S}$, that verifies $x \leq_\oplus e_\oplus$ for every $x \in \mathbb{S} \cup \{e_\oplus\}$. Hence, the $(\mathbb{S} \cup \{e_\oplus\}, \oplus)$structure is an *Abelian idempotent semi-group*[1] (see [BCOQ92]) with $e_\oplus$ as identity element.

In [Duc98], a distorted algebra — the r-algebra — is proposed. This algebra generalizes the Abelian idempotent semi-group, and still allows convergence to terminal configuration of wave-like algorithms.

Definition 16 (r-operator) *The binary operator $\triangleleft$ on $\mathbb{S} \cup \{e_\oplus\}$ is an r-operator if there exists a surjective mapping r called r-mapping, such that it verifies the following conditions: (i) r-associativity: $(x \triangleleft y) \triangleleft r(z) = x \triangleleft (y \triangleleft z)$; (ii) r-commutativity: $r(x) \triangleleft y = r(y) \triangleleft x$; (iii) r-idempotency: $r(x) \triangleleft x = r(x)$ and (iv) right identity element: $\exists e_\triangleleft \in \mathbb{S} \cup \{e_\oplus\}, x \triangleleft e_\triangleleft = x$.*

Definition 17 (Strict idempotency) *An r-operator $\triangleleft$ based on the s-operator $\oplus$ is strictly idempotent if, for any $x \in \mathbb{S} \setminus \{e_\oplus\}$, $x <_\oplus r(x)$ (i.e. $x \leq_\oplus r(x)$ and $r(x) \neq x$).*

For example, the operator $\min_1(x, y) = \min(x, y + 1)$ is a strictly idempotent r-operator on $\mathbb{N} \cup \{+\infty\}$, with $+\infty$ as its identity element. It is based on the *s-operator* min and on the surjective r-mapping $r(x) = x + 1$. Note that although the set $\mathbb{N} \cup \{+\infty\}$ has a greatest element, it is an infinite set, and its greatest element can not be used as an upper bound for a particular algorithm (such as in the works of [AGH90, CG02]).

In [DT01], it is proved that if a strictly idempotent r-operator is executed at every processor in the network, then the system is self-stabilizing for the operation defined by the r-operator. The r-operators defined in [DT01] provide solutions for minimum distance tree and forest, shortest path tree and forest, best transmitters tree and forest, etc. For example, the operator $\min_1$ stabilizes to a minimum distance tree when a single node has 0 as the first term of

---

[1]The prefix *semi* means that the structure cannot be completed to obtain a group, since the law is idempotent.

every $\min_1$ computation and every other node has $e_{\min_1} = +\infty$ as the first term of every $\min_1$ computation.

The algorithm that we provide is self-stabilizing for every r-operator defined in [DT01], yet preserves routes for every message sent to the root when the r-functions of the system are modified after a tree is constructed.

The generic algorithm It is parametrized by an n-ary r-operator $\lhd$. A mapping $\lhd$ from $\mathbb{S} \cup \{e_\oplus\}^n$ into $\mathbb{S} \cup \{e_\oplus\}$ is an *n-ary r-operator* if there exists an *s-operator* $\oplus$ on $\mathbb{S} \cup \{e_\oplus\}$ and $n-1$ homomorphisms (called *r-mappings*) $r_1, \ldots, r_{n-1}$ of $(\mathbb{S} \cup \{e_\oplus\}, \oplus)$ such that $\lhd(x_0, \ldots, x_{n-1}) = x_0 \oplus r_1(x_1) \oplus \cdots \oplus r_{n-1}(x_{n-1})$ for any $x_0, \ldots, x_{n-1}$ in $\mathbb{S} \cup \{e_\oplus\}$.

For our purpose, we associate to each link (i, j) of $G = (V, E)$ a bijective r-function $r_{i,j}$ that is strictly idempotent. Each processor i has the following inputs: *(i)* $\mathcal{N}_i$ is the locally ordered set of neighbors of i, and *(ii)* $r_{i,j}$ is the bijective r-function associated to the edge between processors i and j. Each processor i (except the root) has the following variables: *(i)* $rw_i^\oplus$ (the current value broadcast in i's sub-tree) is an integer, and *(ii)* p_i (the parent of i) is a pointer to a neighbor of i. Each processor i (including the root) has the following variables: *(i)* $w_i^\oplus$ (the weight of i, in the sense of $\oplus$) is an integer, and *(ii)* st_i (the status of i) takes value in $\{\mathsf{P}, \mathsf{N}\}$. Note that $w_i^\oplus$ and $rw_i^\oplus$ are elements of $\mathbb{S} \cup \{e_\oplus\}$.

All processors has the following local macro: $\hat{w}_i^\oplus$ is the best possible value of $w_i^\oplus$ according to the current configuration. Let j be a child of i ($p_j = i$). j is called a *descendant* of i iff $w_i^\oplus <_\oplus w_j^\oplus$. Each processor (except the root) has the following local macro: $\hat{p}_i$ is the best possible parent i according to the current configuration if a such neighbor exists otherwise the value of $\hat{p}$ is $\perp$. $r_{i,j}^{-1}$ is the reverse function of $r_{i,j}$ (since $r_{i,j}$ is bijective, $r_{i,j}^{-1}$ is always defined, and the best parent j should have the neutral status and should have a weight equal to $r_{i,j}^{-1}(\hat{w}_i^\oplus)$), $\mathcal{D}_i$ is the set of the descendants of i in the tree, and $ubw_i^\oplus$ is the upper bound of w_i (in the sense of $\oplus$) in the current configuration (see table below).

$$\hat{w}_i^\oplus \quad \equiv \quad \begin{cases} \oplus_{j \in \mathcal{N}_i}(r_{i,j}(w_j^\oplus)) & \text{if } i \neq r \\ 0 & \text{otherwise} \end{cases}$$

$$\hat{p}_i \quad \equiv \quad \begin{cases} \oplus_{j \in \mathcal{N}_i}(j :: (w_j^\oplus = r_{i,j}^{-1}(\hat{w}_i^\oplus)) \wedge (st_j = \mathsf{N})) & \text{if there exists } j \in \mathcal{N}_i :: \\ & (w_j^\oplus = r_{i,j}^{-1}(\hat{w}_i^\oplus)) \wedge (st_j = \mathsf{N}) \\ \perp & \text{otherwise} \end{cases}$$

$$\mathcal{D}_i \quad \equiv \quad \{j \in \mathcal{N}_i :: p_j = i \wedge w_i^\oplus <_\oplus w_j^\oplus\}$$

$$ubw_i^\oplus \quad \equiv \quad \begin{cases} \oplus_{j \in \mathcal{D}_i} r_{i,j}^{-1}(w_j^\oplus) & \text{if } \mathcal{D}_i \neq \emptyset \\ \infty & \text{otherwise} \end{cases}$$

Each processor (except the root) has the following local predicates: End_PIF_i is used to avoid that a processor increases its weight (i.e. to perform R_3 action) simultaneously with one of its descendants, $Safe\_MOVE_i$ is used to avoid that a processor i takes the status P

(i.e. to perform R_2 action) when it could change its parent, and $Safe\_INC_i$ is verified by i when it needs to increase its weight (see table below).

$$End\_PIF_i \quad \equiv \quad (\forall j \in \mathcal{D}_i, st_j = \mathsf{N})$$

$$Safe\_MOVE_i \quad \equiv \quad \left(\wedge \begin{array}{l} ((\hat{w}_i^{\oplus} <_{\oplus} w_i^{\oplus}) \vee ((w_i^{\oplus} = \hat{w}_i^{\oplus}) \wedge (\hat{p}_i \neq p_i))) \\ (\hat{p}_i \neq \bot) \end{array} \right)$$

$$Safe\_INC_i \quad \equiv \quad \left(\vee \begin{array}{l} ((w_i <_{\oplus} r_{p_i,i}(rw_{p_i}^{\oplus})) \wedge (st_{p_i} = \mathsf{P})) \\ (w_i <_{\oplus} r_{p_i,i}(w_{p_i}^{\oplus})) \end{array} \right)$$

The rules of our generic algorithm are the following:

1. On the root r processor:

   - $R_0 :: (w_r^{\oplus} \neq 0) \vee (st_p \neq \mathsf{N}) \rightarrow$
     $w_r \leftarrow 0; st_r \leftarrow \mathsf{N}$

2. On any other processor i:

   - $R_1 :: (st_i = \mathsf{N}) \wedge Safe\_MOVE_i \rightarrow$
     $w_i^{\oplus} \leftarrow \hat{w}_i^{\oplus}; \ rw_p^{\oplus} \leftarrow w_i^{\oplus}; \ p_i \leftarrow \hat{p}_i$
   - $R_2 :: (st_i = \mathsf{N}) \wedge \neg Safe\_MOVE_i \wedge Safe\_INC_i \rightarrow$
     $rw_i^{\oplus} \leftarrow r_{p_i,i}(rw_{p_i}^{\oplus}); \ st_i \leftarrow \mathsf{P}$
   - $R_3 :: (st_i = \mathsf{P}) \wedge End\_PIF_i \wedge (rw_i^{\oplus} \leq_{\oplus} ubw_i^{\oplus}) \rightarrow$
     $w_i^{\oplus} \leftarrow rw_i^{\oplus}; \ st \leftarrow \mathsf{N}$
   - $R_4 :: (rw_i^{\oplus} <_{\oplus} w_i^{\oplus}) \rightarrow rw_i^{\oplus} \leftarrow w_i^{\oplus}$

We now quickly sketch two possible applications of the generic algorithm. The interested reader can refer to [DT01] for more details. First, to solve the shortest path problem with r-operators, it is sufficient to consider $\mathbb{N}$ as $\mathbb{S}$, $+\infty$ as $e_{\oplus}$, min as $\oplus$, and $x \mapsto x + c_{i,j}$ as r_i^j. Second, in a telecommunication network where some terminals must chose their "best" transmitter, distance is not always the relevant criterium, and it can be interesting to know the transmitter form where there exists a least failure rate path, and to know the path itself. If we consider $[0,1] \cap \mathbb{R}$ as $\mathbb{S}$, 0 as $e_{\oplus}$, max as $\oplus$, and $x \mapsto x \times \tau_i^j$ as r_i^j (where τ_i^j is the reliability rate $- 0 < \tau_i^j < 1$ – of the edge between i and j) our parametrized algorithm ensures that a best transmitter tree is maintained despite transient failures (in a self-stabilizing way) and that once a coherent tree is constructed towards a transmitter, a coherent tree remains event if edge rates continue changing.

Chapter 6

Conclusion

In this paper, we presented a self-stabilizing loop-free routing algorithm that is also *route preserving*. This algorithm also does not require that a bound on the network diameter is known to all processors in the network.

These two key properties make our approach suitable for mobile *ad-hoc* networks, where nodes move often (inducing changes in the diameter of the system and in the edge costs). Unlike previous approaches on self-stabilizing routing and mobile networks, we specifically address the message delivery issue, even in an environment where dynamic changes occur all the time.

Bibliography

[AGH90]   A Arora, MG Gouda, and T Herman. Composite routing protocols. In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pages 70–78, 1990.

[AKM+93] B Awerbuch, S Kutten, Y Mansour, B Patt-Shamir, and G Varghese. Time optimal self-stabilizing synchronization. In *STOC93 Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 652–661, 1993.

[BCOQ92] F Baccelli, G Cohen, G Olsder, and JP Quadrat. Synchronization and linearity, an algebra for discrete event systems. *Series in Probability and Mathematical Statistics*, 1992.

[CG02]   JA Cobb and MG Gouda. Stabilization of general loop-free routing. *Journal of Parallel and Distributed Computing*, 62(5):922–944, 2002.

[Dij74]   EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17(11):643–644, 1974.

[Dol97]   S Dolev. Self-stabilizing routing and related protocols. *Journal of Parallel and Distributed Computing*, 42(2):122–127, 1997.

[DT01]   B Ducourthial and S Tixeuil. Self-stabilization with r-operators. *Distributed Computing*, 14(3):147–162, 2001.

[Duc98]   B Ducourthial. New operators for computing with associative nets. In *Proceedings of the Fifth International Colloquium on Structural Information and Communication Complexity (SIROCCO'98), Amalfi, Italia*, pages 51–65, 1998.

[GS99]   MG Gouda and M Schneider. Stabilization of maximal metric trees. In *Proceedings of the Fourth Workshop on Self-Stabilizing Systems (published in association with ICDCS99 The 19th IEEE International Conference on Distributed Computing Systems)*, pages 10–17. IEEE Computer Society, 1999.

[Per00]   R Perlman. *Interconnexion Networks*. Addison Wesley, 2000.

[Tel94]   G Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[VJ00]   G Varghese and M Jayaram. The fault span of crash failures. *Journal of the Association of the Computing Machinery*, 47(2):244–293, 2000.

RAPPORTS INTERNES AU LRI - ANNEE 2003

N°	Nom	Titre	Nbre de pages	Date parution
1345	FLANDRIN E LI H WEI B	A SUFFICIENT CONDITION FOR PANCYCLABILITY OF GRAPHS	16 PAGES	01/2003
1346	BARTH D BERTHOME P LAFOREST C VIAL S	SOME EULERIAN PARAMETERS ABOUT PERFORMANCES OF A CONVERGENCE ROUTING IN A 2D-MESH NETWORK	30 PAGES	01/2003
1347	FLANDRIN E LI H MARCZYK A WOZNIAK M	A CHVATAL-ERDOS TYPE CONDITION FOR PANCYCLABILITY	12 PAGES	01/2003
1348	AMAR D FLANDRIN E GANCARZEWICZ G WOJDA A P	BIPARTITE GRAPHS WITH EVERY MATCHING IN A CYCLE	26 PAGES	01/2003
1349	FRAIGNIAUD P GAURON P	THE CONTENT-ADDRESSABLE NETWORK D2B	26 PAGES	01/2003
1350	FAIK T SACLE J F	SOME b-CONTINUOUS CLASSES OF GRAPH	14 PAGES	01/2003
1351	FAVARON O HENNING M A	TOTAL DOMINATION IN CLAW-FREE GRAPHS WITH MINIMUM DEGREE TWO	14 PAGES	01/2003