Self-stabilization with r-operators in Unreliable Directed Networks

Sylvie Delaët[a] Bertrand Ducourthial[b] Sébastien Tixeuil[a,c]

[a] Laboratoire de Recherche en Informatique, UMR CNRS 8623,
Université Paris Sud, France
[b] Laboratoire Heudiasyc, UMR CNRS 6599,
Université de Technologie de Compiègne, France,
[c] INRIA Futurs, Équipe Grand Large

Abstract

This paper describes a parameterized distributed algorithm applicable to any directed network. This algorithm tolerates transient faults that corrupt the processors and communication links memory (it is self-stabilizing) as well as intermittent faults (fair loss, reorder, finite duplication of messages) on communication media. A formal proof establishes its correctness for the considered problem. The function parameter of our algorithm can be instantiated to produce distributed algorithms for both fundamental and high level applications, such as shortest path calculus and depth-first-search tree construction. Due to fault resilience properties of our algorithm, the resulting protocols are self-stabilizing at no additional cost.

Keywords: r-operator, distributed systems, self-stabilization, message-passing communications, unreliable communications

Résumé

Cet article décrit un algorithme paramétré applicable à n'importe quelle topologie orientée. Cet algorithme tolère des défaillances transitoires qui corrompent les mémoires des processeurs et des liens de communication (il est auto-stabilisant) ainsi que les défaillances intermittentes (perte équitable, réordonnancement, duplication finie de messages) sur les media de communication. Une preuve formelle établit sa correction pour le problème considéré. La fonction paramètre de notre algorithme peut être instanciée pour produire des algorithmes répartis pour plusieurs applications fondamentales et de haut niveau, comme le calcul des plus courts chemins, et la construction d'un arbre en profondeur. Grâce aux propriétés de recouvrement de fautes de notre algorithme, les protocoles résultants sont auto-stabilisants sans surcoût.

Mots-clef: r-opérateur, systèmes distribués, auto-stabilisation, communications par passage de messages, communications non fiables.

Chapter 1

Introduction

Robustness is one of the most important requirements of modern distributed systems. Two approaches are possible to achieve fault-tolerance: on the one hand, robust systems use redundancy to mask the effect of faults, on the other hand, self-stabilizing systems may temporarily exhibit an abnormal behavior, but must recover correct behavior within finite time.

At the contrary of fault-tolerance, self-stabilization does not make any restriction on the subset of the system that is hit by the failure. Since its introduction by Dijkstra (see [13]), a growing number of self-stabilizing algorithms solving different problems have been presented (see [14]). In particular, several recent publications prove that being able to start from any arbitrary configuration is desirable as a property of fault tolerance. For example, [26] shows that processor crashed and restarted may lead a system to an arbitrary global state, from which a self-stabilizing algorithm is able to recover.

1.1 Related Work

Historically, research in self-stabilization over general networks has mostly covered undirected networks where bidirectional communication is feasible (the Update protocol of [16], or the algorithms presented in [2, 17]). Recently, studying self-stabilizing solutions for directed networks received a lot of attention ([11, 19, 12, 20]), due to the two main following reasons:

1. If a bidirectional network is not available, then self-stabilizing data link protocols (that are acknowledgment based, such as those presented in [1]) can not be used to transform any of those works written for shared memory systems so that they perform in unreliable message passing environments.

2. In directed networks, it is generally easy to maintain the set of input neighbors (by checking who has "recently" sent a message), but it is very difficult (if not impossible) to maintain the set of output neighbors (in a satellite or a sensor network, a transmitter is generally not aware of who is listening to the information it communicates). As a result, a large part of self-stabilizing algorithms, that use implicit neighborhood knowledge (*e.g.* [5, 6]) to compare one node state with those of its neighbors and check for consistency can not be used in directed networks.

Several algorithms are self-stabilizing and tolerate a limited amount of processor crash failures (see [3, 21, 24, 9]). However, they are studied in a communication model that is almost reliable (links are only subject to transient failures). In [8], the authors consider the case of systems subject to crash failures for processors and intermittent failures for links (only the loss case is considered). However, in their approach, bidirectional communication link are assumed to provide a lower level communication protocol that is reliable enough for their purpose. To some extent, topology changes can be considered as permanent failures on links. In this context, Super-stabilizing and Snap-stabilizing protocols (introduced in [16] and [10], respectively) are self-stabilizing protocols that also tolerate limited topology changes. In [1], Afek and Brown consider self-stabilizing algorithms along with lossy communication links, but they assume bidirectional communications in order to build an underlying self-stabilizing data-link protocol. The construction of wait-free objects in a self-stabilizing setting is considered in [22]. Finally, [12] solves the census problem and is self-stabilizing yet supporting fair loss, finite duplication, and desequencing of messages.

1.2 Our Contribution

In this paper, we concentrate on providing a generic solution to silent tasks (see [15]) in a self-stabilizing way on a truly general directed network, where no hypothesis are made about the strong connectivity or the presence of cycles. As in [4], our solution is by giving a condition on the distributed algorithm. However, in [4], the condition is given in terms of global system property, while our condition is independent of the task to be solved, and is only determined by the algebraic properties of the function computed locally by the algorithm. As in [19, 20], our solution does not require any knowledge about the network: no size, diameter, maximum degree are needed. In addition, we support message passing communications with the same communications hazards as in [12]: fair loss, finite duplication, and reordering of messages.

In more details, we provide a parametrized algorithm that can be instantiated with a local function. Our parameterized algorithm enables a set of silent tasks to be solved self-stabilizingly, provided that these tasks can be expressed through local calculus operations called r-operators that operate over a set $\mathbb{S}$. The r-operators are general enough to permit applications such as shortest path calculus and depth-first-search tree construction, to be solved on arbitrary graphs while remaining self-stabilizing.

In addition, since our approach is condition based, there is no additional layer used to make an algorithm that satisfies this condition tolerant to transient failures. In fact, when no transient faults appear in the system, the performance suffers no overhead. In the following table, we capture the key differences between our protocol and the aforementioned related solutions in general directed networks regarding the following criteria: communication (processors communicate through shared registers, which is costly emulated in message passing systems, or simply by exchanging messages), atomicity (composite atomicity is when a node may read the shared memory of all of its neighbors in one atomic step, which is more expensive than read/write atomicity where only a single read or write action is atomic, which is in turn costly emulated in message passing systems), reliability (communication between nodes can be assumed to be reliable, which involves more resources in a faulty environment), and algorithm nature (a specific algorithm solves one particular problem, while a generic one can be instantiated for many purposes).

Reference	Communication	Atomicity	Reliability	Algorithm
[11]	message passing	send/receive atomicity	reliable	specific (routing)
[12]	message passing	send/receive atomicity	unreliable	specific (census)
[20]	shared memory	composite atomicity	reliable	generic (partial order on $\mathbb{S}$)
[19]	shared memory	read/write atomicity	reliable	generic (total order on $\mathbb{S}$)
This paper	message passing	send/receive atomicity	unreliable	generic (total order on $\mathbb{S}$)

1.3 Outline

Chapter 2 presents a model for distributed systems we consider, as well as convenient notations used in the rest of the paper. Chapter 3 describes our self-stabilizing parameterized algorithm general directed networks. Due to lack of space, the correctness is proved in Chapter 4. Concluding remarks are proposed in Chapter 5.

Chapter 2

Model

In this section, we give general definitions on message passing self-stabilizing distributed systems.

2.1 Distributed system

Processors. A *processor* is a sequential deterministic machine that uses a local memory, a local algorithm and input/output capabilities. Such a processor executes its local algorithm, that modifies the state of its memory, and sends/receives messages using the communication ports.

Links. An *unidirectional communication link* transmits messages from an origin processor o to a destination processor d. The link is interacting with one input port of d and one output port of o.

Depending on the way messages are handled by a communication link, several properties can be defined on a link. A complete formalization of these properties is proposed in [23]. We only enumerate those that are related to our algorithm. There is a *fair loss* when, infinitely many messages being emitted by o, infinitely many messages are received by d. There is *finite duplication* when every message emitted by o may be received by d a finite (yet unbounded) number of times. There is *reordering* when messages emitted by o may be received by d in a different order than that they were emitted. There is *eventual delivery* if any message that is not lost is eventually received.

Distributed system. A *distributed system* is a 2-tuple $\mathcal{S} = (\mathcal{P}, \mathcal{L})$ where $\mathcal{P}$ is the set of processors and $\mathcal{L}$ is the set of communication links. Such a system is modeled by a *directed graph* (also called *digraph*) $G = (V, E)$, defined by a set of vertices V and a set E of edges (v_1, v_2), which are ordered pairs of vertices of V $(v_1, v_2 \in V)$. Each vertex u in V represents a processor P_u of system $\mathcal{S}$. Each edge (u, v) in E represents a communication link from P_u to P_v in $\mathcal{S}$. In the remaining of the paper, we use indifferently processors, nodes, and vertices to denote processors, and links and edges to denote communication links.

Graph definitions. The *in-degree* of a vertex v of G, denoted by δv is equal to the number of vertices u such that the edge (u, v) is in E. The incoming edges of each vertex v of G are indexed from 1 to δv. A *directed path* P_{v_0,v_k} in a digraph $G(V, E)$ is an ordered list of vertices $v_0, v_1, \ldots, v_k \in V$ such that, for any $i \in \{0, \ldots, k-1\}$, (v_i, v_{i+1}) is an edge of E (*i.e.*, $(v_i, v_{i+1}) \in E$). The *length* of this path is k. If each v_i is unique in the path, the path is *elementary*. The set of all elementary paths from a vertex u to another vertex v is denoted by $\mathcal{X}_{u,v}$. A *cycle* is a directed path P_{v_0,v_k} where $v_0 = v_k$. The *distance* between two vertices u, v of a digraph G (denoted by $\mathrm{d}_G(u, v)$, or by $\mathrm{d}(u, v)$ when G is not ambiguous) is the minimum of the lengths of all directed paths from u to v (assuming there exists at least one such path). The *diameter* of a digraph G is the maximum of the distances between all couples of vertices in G between which a distance is defined. Finally, we denote as Γ_v^- (resp. Γ_v^+) the set of ancestors (resp. successors) of a vertex $v \in V$, that is the set of all vertices $u \in V$ such that there exists a path starting at u (resp. v) and ending at v (resp. u). The ancestors (resp. successors) u of v verifying $\mathrm{d}_G(u, v) = 1$ (resp. $\mathrm{d}_G(v, u) = 1$)) are called *direct-ancestors* (resp. *direct-successors*) and their set is denoted Γ_v^{-1} (resp. Γ_v^{+1}).

Configurations. The state of a processor can be reduced to the state of its local memory while the state of a communication link can be reduced to its contents. So the global system state, called a *system configuration* (or simply *configuration*) and generally denoted c, is the union of (i) the states of memories of processors of $\mathcal{P}$ and (ii) the contents of communication links of $\mathcal{L}$. The set of configurations is denoted by $\mathcal{C}$. The part of informations in a configuration $c \in \mathcal{C}$ related to the processors of $\mathcal{P}$ is denoted by $c_{|\mathcal{P}}$.

2.2 Distributed algorithm

Distributed algorithm. A *distributed algorithm* Alg (or *protocol*) is a collection of local algorithms. A distributed system $\mathcal{S}$ executes Alg if every processor of $\mathcal{S}$ executes a local algorithm of Alg.

A distributed system that executes a protocol Alg is not fixed: it passes from a configuration to another when a processor executes an instruction of its local algorithm or when a communication link delivers a message to its destination. The set of all such actions that can be performed in the distributed system $\mathcal{S}$ is denoted $\mathcal{A}$.

Executions. Starting from an *initial configuration* c_1, an *execution* $e_{c_1} = c_1, a_1, c_2, a_2, \ldots$ is a maximal alternating sequence of configurations of $\mathcal{C}$ and actions of $\mathcal{A}$ such that, for any positive integer i, the transition from configuration c_i to configuration c_{i+1} is done through execution of action a_i. The notations $\mathcal{E}_c$, $\mathcal{E}_C$ and $\mathcal{E}$ denote respectively the set of all executions starting (i) from the initial configuration c, (ii) from any configuration $c \in C \subset \mathcal{C}$, or (iii) from any configuration of $\mathcal{C}$ ($\mathcal{E}_{\mathcal{C}} = \mathcal{E}$). The ordered list $c_1, c_2, \ldots \in \mathcal{C}$ of the configurations of an execution $e = c_1, a_1, c_2, a_2 \ldots$ is denoted by $e_{|\mathcal{C}}$. In the rest of this paper, we adopt the following convention: if $c_i \in e_{|\mathcal{C}}$ appears before $c_j \in e_{|\mathcal{C}}$, then $i < j$.

Specification. Distributed algorithms resolve either static tasks (*e.g.*, distance computation) or dynamic tasks (*e.g.*, token circulation). The aim of static tasks is to compute a global result, which means that after a running time, processors always produce the same output (*e.g.*, the distance from a source). This paper focus on static tasks.

By definition, a static task is characterized by some final processor's outputs o, called *legitimate outputs*. A *legitimate configuration* c for this task satisfies $c_{|\mathcal{P}} = o$. A distributed protocol designed for solving a given static task is correct if the distributed system $\mathcal{S}$ running this protocol reaches in finite time a legitimate configuration for this task.

2.3 Self-stabilizing protocols

Robustness against transient failures. Self-stabilizing algorithms are designed for distributed systems where some *transient failures* could happen. A transient failure spontaneously changes a value in a processor memory or in a message in transit.

A self-stabilizing algorithm does not always satisfy its specification. However, it seeks to reach a configuration from which any execution verifies its specification. For algorithms that solve static tasks, this means that, starting from a configuration where processor's outputs are correct, the system may reach a configuration where processor's output is no more correct (*e.g.* assuming a transient failure occurs somewhere in the system). However, if the algorithm is self-stabilizing, it eventually reaches again a correct configuration without human intervention.

Self-stabilization. We now define precisely a self-stabilizing distributed protocol. A set of configurations $C \subset \mathcal{C}$ is *closed* if, for any $c \in C$, any possible execution $e_c \in \mathcal{E}_c$ of system $\mathcal{S}$ whose c is initial configuration only contains configurations in C. A set of configurations $C_2 \subset \mathcal{C}$ is an *attractor* for a set of configurations $C_1 \subset \mathcal{C}$ if, for any $c \in C_1$ and any execution $e_c \in \mathcal{E}_c$, the execution e_c contains a configuration of C_2. Let $C \subset \mathcal{C}$ be a non-empty set of configurations. A distributed system $\mathcal{S}$ is *C-stabilizing* if and only if C is a closed attractor for $\mathcal{C}$: any execution e of $\mathcal{E}$ contains a configuration c of C, and any further configurations in e reached after c remains in C.

Finally, let consider a static task for the distributed system $\mathcal{S}$, and let $L \subset \mathcal{C}$ be the set of its legitimate configurations. A distributed protocol designed for solving this static task is *self-stabilizing* if the distributed system $\mathcal{S}$ running this protocol is L-stabilizing.

Chapter 3

Parametric message passing $\mathcal{PA}$-MP algorithm

In this section, we first describe the distributed system we consider before defining the $\mathcal{PA}$-MP parametrized algorithm. We then introduce the r-operators, that are used as parameters.

3.1 System

Let $\mathcal{S} = (\mathcal{P}, \mathcal{L})$ be the distributed system we consider in the following. The associated graph composed of processors of $\mathcal{P}$ and communications links of $\mathcal{L}$ is fixed, directed and unknown to the processors of $\mathcal{P}$.

Each processor v of $\mathcal{P}$ owns an incoming memory denoted as $\mathtt{IN}_v$, which is supposed to be unalterable; this can be implemented by a ROM memory (*e.g.*, EPROM), or a memory that is regularly reloaded by any external process (human interface, captor, other independent algorithm, *etc.*). The value of this memory (that will never change) is called *initialization value*. Moreover, for each link starting at processor $u \in \mathcal{P}$ and ending at processor v corresponds an incoming memory $\mathtt{IN}_v^u$ in v, which is used by v to store incoming messages sent by u. In addition, processor v owns an output memory denoted by $\mathtt{OUT}_v$. All these memories are private, and can only be read or written by v (note that v only reads $\mathtt{IN}_v$, and only writes $\mathtt{OUT}_v$). In the following, we identify the name of a memory with the value it contains. In the same way, a message is considered as equivalent to its value.

Processor v performs a calculation by applying an operator $\triangleleft$ (see § 3.3) on all its incoming memories, and stores the result in its output memory $\mathtt{OUT}_v$.

3.2 Algorithm

In [19] is defined a Parameterized distributed Algorithm (denoted as $\mathcal{PA}$), and proved that it is self-stabilizing when $\triangleleft$ is a strictly idempotent r-operator (see § 3.3). This algorithm uses shared registers to permit communication between neighboring processors. In this paper, we design a similar parametrized distributed protocol for Message Passing systems (denoted as

$\mathcal{PA}$-MP). This protocol is composed of one local parametrized algorithm per processor v of $\mathcal{P}$, denoted by $\mathcal{PA}$-MP$|_{\lhd_v}$, where $\lhd_v$ is an operator used as a parameter (parameters could be slightly different on each processor, see Hypothesis 2). This local algorithm calls three helper functions:

$\mathtt{Store}_v(m,\ u)$:	stores in the local register $\mathtt{IN}_v^u$ the contents of the message m
$\mathtt{Evaluate}_v(\lhd_v)$:	stores in the local register $\mathtt{OUT}_v$ the result of the local computation $\lhd_v(\mathtt{IN}_v, \mathtt{IN}_v^{u_1}, \ldots, \mathtt{IN}_v^{u_k})$ where $u_1, \ldots, u_k$ are direct ancestors of v $(\in \Gamma_v^{-1})$
$\mathtt{Forward}_v$:	for each processor $w \in \Gamma_v^{+1}$, send $\mathtt{OUT}_v$ to w.

The local algorithm $\mathcal{PA}$-MP$|_{\lhd}$ on processor v is composed of two *guarded actions*, which are sets of instructions (actions) executed when a pre-condition (guard) is fulfilled (see Figure 3.1).

$\mathcal{R}_1$ <u>`Upon receipt of a message` m `send by` u</u>:
```
        if m ≠ IN_v^u , then
            Store_v(m, u)
            Evaluate_v(◁_v)
            Forward_v
        end if
```
$\mathcal{R}_2$ <u>`Upon timeout expired`</u>:
```
        Evaluate_v(◁_v)
        Forward_v
        reset the timeout
```

Figure 3.1: Local algorithm $\mathcal{PA}$-MP$|_{\lhd_v}$ on processor v.

3.3 r-operators

Following work of Tel concerning wave algorithms (see [25]), the distributed protocol described above terminates when each $\mathcal{PA}$-MP local parametric algorithm is instantiated by an infimum over the set of inputs $\mathbb{S}$. An *infimum* (hereby called an *s-operator*) $\oplus$ over a set $\mathbb{S}$ is an associative, commutative and idempotent binary operator. Such an operator defines a partial order relation $\preceq_\oplus$ over the set $\mathbb{S}$ by

$$x \preceq_\oplus y \quad \text{if and only if} \quad x \oplus y = x \tag{3.1}$$

and then a strict order relation $\prec_\oplus$ by $x \prec_\oplus y$ if and only if $x \preceq_\oplus y$ and $x \neq y$. It is generally assumed that there exists a greatest element on $\mathbb{S}$, denoted by $e_\oplus$, and verifying $x \preceq_\oplus e_\oplus$ for every $x \in \mathbb{S}$. If necessary, this element can be added to $\mathbb{S}$. In the following, we assume that an s-operator admits such an element in its definition of set $\mathbb{S}$. Hence, the $(\mathbb{S}, \oplus)$ structure is an

Abelian idempotent semi-group[1] (see [7]) with $e_\oplus$ as identity element. When parameterized by such an s-operator $\oplus$, the $\mathcal{PA}$-MP parametric local algorithm converges [18]. However, some counter examples show that it is not self-stabilizing [19], that is, it cannot recover after some transient failures. The following lemma will be used in the following.

Lemma 1 *For all $x, y, z \in \mathbb{S}$, if $x \oplus y = z$ then $z \preceq_\oplus x$ and $z \preceq_\oplus y$.*

In [18], a distorted algebra — the r-algebra — is proposed. This algebra generalizes the Abelian idempotent semi-group, and still allows convergence of wave-like algorithms.

Definition 1 *The binary operator $\triangleleft$ on $\mathbb{S}$ is an* r-operator *if there exists a surjective mapping r called r-mapping, such that the following conditions are fulfilled: **(i)** r-associativity: $\forall x, y, z \in \mathbb{S}, (x \triangleleft y) \triangleleft r(z) = x \triangleleft (y \triangleleft z)$; **(ii)** r-commutativity: $\forall x, y \in \mathbb{S}, r(x) \triangleleft y = r(y) \triangleleft x$; **(iii)** r-idempotency: $\forall x \in \mathbb{S}, r(x) \triangleleft x = r(x)$ and **(iv)** right identity element: $\exists e_\triangleleft \in \mathbb{S}, x \triangleleft e_\triangleleft = x$.*

Given an r-operator $\triangleleft$, one can show that the r-mapping r is unique, and is an homomorphism of $(\mathbb{S}, \triangleleft)$. Moreover, the r-operator defines an s-operator on $\mathbb{S}$ by $x \triangleleft y = x \oplus r(y)$, and $e_\oplus = e_\triangleleft$. We have: $r(e_\oplus) = e_\oplus$. It is straightforward that an r-operator with the identity mapping as r-mapping is an s-operator. Since r is an homomorphism, we have the following lemma.

Lemma 2 *For all $x, y \in \mathbb{S}$, if $x \preceq_\oplus y$, then $r(x) \preceq_\oplus r(y)$.*

If no fault appears in the distributed system $\mathcal{S}$, our $\mathcal{PA}$-MP algorithm stabilizes when it is parameterized by any idempotent r-operator $\triangleleft$. Idempotent r-operators verify (i) $r(e_\oplus) = e_\oplus$, and (ii) for any $x \in \mathbb{S}$, $x \preceq_\oplus r(x)$. This last property leads to the following definition.

Definition 2 *An r-operator $\triangleleft$ is* strictly idempotent *if, for any $x \in \mathbb{S} \setminus \{e_\oplus\}$, we have $x \prec_\oplus r(x)$.*

For example, the operator $\text{minc}(x, y) = \min(x, y + 1)$ 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 $r(x) = x + 1$. Such an operator can also be defined on the finite set $\{0, 1, \ldots, 255\}$. In that case, the r-mapping is defined by $r(x) = x + 1$ for $x \in \{0, \ldots, 254\}$ and $r(255) = 255$.

Finally, binary r-operators can be extended to accept any number of arguments. This is useful for our algorithm because a processor computes a result with one value per direct ancestor plus its own initialization value. An *n-ary r-operator* $\triangleleft$ consists in $n - 1$ binary r-operators based on the same s-operator, an we have, for any $x_0, \ldots, x_{n-1}$ in $\mathbb{S}$, $\triangleleft(x_0, \ldots, x_{n-1}) = x_0 \oplus r_1(x_1) \oplus \cdots \oplus r_{n-1}(x_{n-1})$. If all of these binary r-operators are (strictly) idempotent, the resulting n-ary r-operator is said (strictly) idempotent.

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

3.4 Hypotheses

In this section, we formalize some hypotheses, introduce some notations, and give basic lemma that will be used for the proofs.

Hypothesis 1 *In the distributed system $\mathcal{S}$, links may (fairly) loose, (finitely) duplicate, and (arbitrarily) reorder messages that are sent by neighboring processors. However, any message sent by u on the link (u, v) that is not lost is eventually received by v.*

The following lemma is immediate.

Lemma 3 *Let consider a communication link $(u, v) \in \mathcal{L}$. If the origin node u keeps sending the same message infinitely often, then this message is eventually received by the destination node v.*

Hypothesis 2 *In the distributed system $\mathcal{S}$ running the $\mathcal{PA}$-MP algorithm, any processor v runs the local algorithm defined in Figure 3.1 and parametrized by a strictly idempotent $(\delta v+1)$-ary r-operator. Moreover, all these r-operators are defined on the same set $\mathbb{S}$, and are based on the same s-operator $\oplus$, with $e_\oplus$ their common identity element.*

The following lemma is a direct application of Hypothesis 2, Definition 1, and `Evaluate` function:

Lemma 4 *Let $\triangleleft_v$ be the r-operator used by processor v. Then the computation of the $\texttt{Evaluate}_v(\triangleleft_v)$ function can be rewritten as: $\triangleleft_v(\texttt{IN}_v, \texttt{IN}_v^{u_1}, \ldots, \texttt{IN}_v^{u_k}) = \texttt{IN}_v \oplus r_v^{u_1}(\texttt{IN}_v^{u_1}) \oplus \cdots \oplus r_v^{u_k}(\texttt{IN}_v^{u_k})$.*

Hence, there is one r-mapping per communication link. We now define the composition of these mappings along a path.

Definition 3 *Let $P_{u_0,u_k} \in \mathcal{X}_{u_0,u_k}$ be a path from processor u_0 to processor u_k, composed of the edges (u_i, u_{i+1}) $(0 \leq i < k)$. Let r_{i+1}^i, $0 \leq i < k$, be the r-mapping associated to the link (u_i, u_{i+1}). The* r-path-mapping *of P_{u_0,u_k}, denoted by $r_{P_{u_0,u_k}}$, is defined by the composition of the r-mappings r_{i+1}^i, for $0 \leq i < k$: $r_{P_{u_0,u_k}} = r_k^{k-1} \circ \cdots \circ r_1^0$.*

Hypothesis 3 *The order relation $\preceq_\oplus$ is a total order relation: $\forall x, y \in \mathbb{S}$, either $x \preceq_\oplus y$ or $y \preceq_\oplus x$.*

Hypothesis 4 *The set $\mathbb{S}$ is either finite, or any strictly increasing infinite sequence of values of $\mathbb{S}$ is unbounded (except by $e_\oplus$).*

This hypothesis specifies that the values used in the distributed system $\mathcal{S}$ can be, for instance, integers but not reals. Note that truncated reals (as in any computer implementation) are also convenient. Hypotheses 2 and 4 give the following lemma:

Lemma 5 *The set $\mathbb{S}$ is either finite or any r-mapping r in $\mathcal{S}$ verifies: $\forall x \in \mathbb{S} \setminus \{e_\oplus\}, r(x) \prec_\oplus e_\oplus$.*

Hypothesis 5 *Each processor v admits at least one ancestor $u \in \Gamma_v^-$ such that $\mathtt{IN}_u \neq e_\oplus$, u is called a* non-null *processor.*

In the following, we denote by $\widehat{\mathtt{OUT}}_v$ the legitimate output of processor v. Moreover, for any processor v, any ancestor u of v and any configuration c, we denote by $\mathtt{OUT}_v(c)$ and $\mathtt{IN}_v^u(c)$ the value of the memories $\mathtt{OUT}_v$ and $\mathtt{IN}_v^u$ in the configuration c.

3.5 Problem specification

Our protocol is dedicated to static tasks. Such tasks (*e.g.*, the distance computation from a processor u) are defined by one output per processor v (*e.g.*, the distance from u to v), which is the legitimate output of v. With our $\mathcal{PA}$-MP algorithm, this means that, after finite time, each processor $v \in \mathcal{P}$ should contain this output (*e.g.*, $d(u, v)$) in its outgoing memory $\mathtt{OUT}_v$. To solve static tasks with the $\mathcal{PA}$-MP distributed algorithm, one must use an operator as parameter (*e.g.*, minc for distance computation) such that the distributed system $\mathcal{S}$ reaches the legitimate configurations and do not leave them thereafter (*i.e.*, any processor reaches and then conserves its legitimate output). In this paper, we prove that if the operator used to parameterize the $\mathcal{PA}$-MP distributed algorithm, then it is self-stabilizing, according to the hypotheses of § 3.4.

Let us define the legitimate outputs of the processors using the r-operators that parameterize the $\mathcal{PA}$-MP algorithm. For instance, to solve the distance computation problem, we state $\mathbb{S} = \mathbb{N} \cup \{+\infty\}$, and each local algorithm is parametrized by the minc r-operator (see § 3.3). All processors v verify $\mathtt{IN}_v = +\infty$ except a non null processor u verifying $\mathtt{IN}_u = 0$ (0 is absorbing while $+\infty$ is the identity element for minc). Each r-path-mapping adds its length to its argument (*i.e.*, $r_P(x) = x + \text{length}(P)$), and we have:

$$d(u, v) = \min\left(\mathtt{IN}_v, \min_{w \in \Gamma_v^-, P_{w,v} \in \mathcal{X}_{w,v}} \left\{ r_{P_{w,v}}(\mathtt{IN}_w) \right\}\right)$$

We now define the legitimate output of a processor v in the general case.

Definition 4 (Legitimate output) *The* legitimate output *of processor v is:*

$$\widehat{\mathtt{OUT}}_v = \mathtt{IN}_v \oplus \bigoplus_{u \in \Gamma_v^-, P_{u,v} \in \mathcal{X}_{u,v}} r_{P_{u,v}}(\mathtt{IN}_u)$$

The following lemma is given by Lemma 5, Hypothesis 5 and Definition 4.

Lemma 6 *The set $\mathbb{S}$ is either finite or any processor $v \in \mathcal{P}$ verifies: $\widehat{\mathtt{OUT}}_v \prec_\oplus e_\oplus$.*

As defined in Chapter 2, the set of legitimate configurations $L \subset \mathcal{C}$ of the protocol **Alg** is given by: for any configuration $c \in L$, for any processor $v \in \mathcal{P}$, $\mathtt{OUT}_v(c) = \widehat{\mathtt{OUT}}_v$.

Theorem 1 *Algorithm $\mathcal{PA}$-MP parametrized by any strictly idempotent r-operator self-stabilizes to a legitimate output at every processor $v \in \mathcal{P}$ despite fair loss, finite duplication and reordering of messages.*

Chapter 4

Correctness

This section is divided into five parts. First, we prove that eventually the output of each processors is updated using its inputs. Second, we show that eventually each received message was sent in the past. Third, we prove that each processor's output is upper bounded. Fourth, we prove that each processor eventually reaches its legitimate value. Finally, we present complexity results regarding our distributed protocol.

4.1 Outputs eventually result from computations

We begin by defining some predicates on configurations.

Definition 5 *Let P_{0a}, P_{0b} and P_{0c} be predicates on configurations $c \in \mathcal{C}$:*

$$\begin{array}{lcl} P_{0a}(c) & \equiv & \forall v \in \mathcal{P}, \quad \mathtt{OUT}_v(c) \preceq_{\oplus} \mathtt{IN}_v \\ P_{0b}(c) & \equiv & \forall v \in \mathcal{P}, \forall u \in \Gamma_v^{-1}, \quad \mathtt{OUT}_v(c) \preceq_{\oplus} r_v^u(\mathtt{IN}_v^u(c)) \\ P_{0c}(c) & \equiv & \forall v \in \mathcal{P}, \forall u \in \Gamma_v^{-1}, \quad \mathtt{OUT}_v(c) = \mathtt{IN}_v \vee \mathtt{OUT}_v(c) = r_v^u(\mathtt{IN}_v^u(c)) \end{array}$$

Now, the set $Q_0 \subset \mathcal{E}$ includes executions where processors eventually update their output. Every execution e of Q_0 reaches a configuration c_{i_0} such that any subsequent configuration c_j satisfies Predicates P_{0a}, P_{0b} and P_{0c}.

Definition 6 *Let $Q_0 \subset \mathcal{E}$ be the set of executions such that:*

$$\forall e \in Q_0, \quad \exists c_{i_0} \in e_{|\mathcal{C}}, \forall c_j \in e_{|\mathcal{C}} \text{ with } i_0 \leq j, \qquad P_{0a}(c_j) \wedge P_{0b}(c_j) \wedge P_{0c}(c_j)$$

We now prove that, thanks to weak fairness hypothesis, any execution of $\mathcal{E}$ is in Q_0.

Lemma 7 *Every execution of the $\mathcal{PA}$-MP algorithm in the distributed system $\mathcal{S}$ is in Q_0.*

Proof: Let $e \in \mathcal{E}$ be an execution. By weak fairness, every processor $v \in \mathcal{P}$ eventually executes a rule. By definition of $\mathcal{PA}$-MP (see Figure 3.1), any execution of either rule at some node v processes $\texttt{Evaluate}_v(\lhd_v)$. Then, for any processor $v \in \mathcal{P}$, there exists a configuration $c_{i_v} \in e_{|\mathcal{C}}$ where processor v satisfies $\texttt{OUT}_v(c_{i_v}) = \lhd_v(\texttt{IN}_v, \texttt{IN}_v^{u_1}(c_{i_v}), \ldots, \texttt{IN}_v^{u_{\delta v}}(c_{i_v}))$.

By Lemma 4, we have $\texttt{OUT}_v(c_{i_v}) = \texttt{IN}_v \oplus r_v^{u_1}(\texttt{IN}_v^{u_1}(c_{i_v})) \oplus \cdots \oplus r_v^{u_{\delta v}}(\texttt{IN}_v^{u_{\delta v}}(c_{i_v}))$. Then, by Lemma 1, we have $\texttt{OUT}_v(c_{i_v}) \preceq_\oplus \texttt{IN}_v$ and $\texttt{OUT}_v(c_{i_v}) \preceq_\oplus r_v^u(\texttt{IN}_v^u)$ for any direct-ancestor u of v. Hence, both $P_{0a}(c_{i_v})$ and $P_{0b}(c_{i_v})$ hold. Now, since $\preceq_\oplus$ defines a total order relation (Hypothesis 3), either $\texttt{OUT}_v(c_{i_v}) = \texttt{IN}_v$ or $\texttt{OUT}_v(c_{i_v}) = r_v^u(\texttt{IN}_v^u(c_{i_v}))$ for at least one ancestor u of v. This gives $P_{0c}(c_{i_0})$ with $i_0 = \max_{v \in \mathcal{P}} i_v$.

Since any action of v executed upon receipt of a message or upon timeout expiration calls $\texttt{Evaluate}$, any subsequent configuration satisfies Predicates P_{0a} to P_{0c}. □

4.2 Eventually, received messages were previously sent

We define the set Q_1 as the subset of executions $\mathcal{E}$ for which any received value has actually be sent in the past. All executions e of Q_1 reach a configuration c_{i_1} such that, for any subsequent configuration c_j and any communication link (u, v), there exists a configuration $c_{j_{uv}}$ in which v sent the value contained in $\texttt{IN}_v^u$ in configuration c_j.

Definition 7 *Let $Q_1 \subset \mathcal{E}$ be the set of executions that satisfy:*

$$\forall e \in Q_1, \quad \exists c_{i_1} \in e_{|\mathcal{C}} \begin{cases} \forall c_j \in e_{|\mathcal{C}} \text{ with } i_1 \leq j, \forall (u,v) \in \mathcal{L}, \\ \exists c_{j_{uv}} \in e_{|\mathcal{C}} \text{ with } j_{uv} \leq j, \quad \texttt{OUT}_u(c_j) = \texttt{IN}_v^u(c_{j_{uv}}) \end{cases}$$

We now prove that, thanks to Hypothesis 1 related to the properties of the communications links, any execution is in Q_1.

Lemma 8 *Every execution of the $\mathcal{PA}$-MP algorithm in the distributed system $\mathcal{S}$ is in Q_1.*

Proof: Let $e \in \mathcal{E}$ be an execution, and consider two processors u and v such that (u, v) is a communication link of $\mathcal{L}$. By definition of $\mathcal{PA}$-MP, processor v sends the value of its $\texttt{OUT}_v$ variable infinitely often to each of its direct successors. By Hypothesis 1, every message that is not lost is eventually delivered. Moreover, every message may be duplicated only a finite number of times. It follows that, after a finite amount of time, only messages that were sent by v are received by every of its direct successors. Hence, there exists a configuration $c_j \in e_{|\mathcal{C}}$ where the incoming value in $\texttt{IN}_v^u$ has actually been sent by u in a previous configuration $c_{j_{uv}}$:

$$\texttt{IN}_v^u(c_j) = \texttt{OUT}_u(c_{j_{uv}}) \text{ with } j_{uv} \leq j \tag{4.1}$$

After all initial erroneous messages between u and v have been received (including duplicates), and after a configuration where the above property holds, this property remains thereafter on this link. Since all links conform to the same hypotheses, there exists a configuration $c_{i_1} \in e_{|\mathcal{C}}$ where the property holds (and remains so thereafter) for any communication link. We conclude that $e \in Q_1$. □

Note that this lemma does not indicate that any sent value is eventually received. Indeed, it may happen that a message is lost while traversing a link, and the variable it was built with is erased by a new value. Then, any re-sending would not provide the original value, that would not be received again. We now generalize the notation we introduced in the previous proof.

Definition 8 *Let us consider an incoming value* $\mathtt{IN}_v^u(c_j)$ *on processor* v *in the configuration* c_j*. Then we denote by* $c_{j_{uv}}$ *the configuration in which the value* $\mathtt{IN}_v^u(c_j)$ *has been sent by* u*, provided that this configuration exists.*

The previous lemma indicates that, for any execution $e \in \mathcal{E}$, there exists a configuration c_{i_1} from which $c_{j_{uv}}$ exists for any subsequent configuration c_j ($i_1 \leq j$), and any communication link (u, v). However, as captured in Figure 4.1, the definition of Q_1 gives no guarantees about $c_{j_{uv}}$ appearing after configuration c_{i_1} (that is $i_1 \leq j_{uv}$).

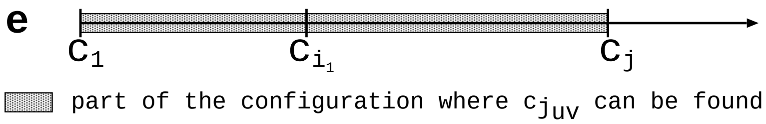

Figure 4.1: According to Q_1, configuration $c_{j_{uv}}$ exists but could appear before c_{i_1}.

We now introduce additional sets of executions. The following definition, illustrated in Figure 4.2, indicates that, for any execution in Q_{1b}, from a given configuration $c_{i_{1b}}$, any given configuration c_i admits a configuration $c_{i'}$ such that any configuration $c_{j_{uv}}$ (with $i' \leq j$) appeared after c_i (*i.e.*, $i \leq j_{uv}$).

Definition 9 *Let* $Q_{1b} \subset \mathcal{E}$ *be the set of executions that satisfy:*

$$\forall e \in Q_{1b}, \quad \exists c_{i_{1b}} \in e_{|\mathcal{C}} \begin{cases} \forall c_i \in e_{|\mathcal{C}} \textit{ with } i_{1b} \leq i, \exists c_{i'} \in e_{|\mathcal{C}} \textit{ with } i \leq i', \\ \forall c_j \in e_{|\mathcal{C}} \textit{ with } i' \leq j, \forall (u,v) \in \mathcal{L}, \qquad c_{j_{uv}} \in e_{|\mathcal{C}} \quad \wedge \quad i \leq j_{uv} \leq j \end{cases}$$

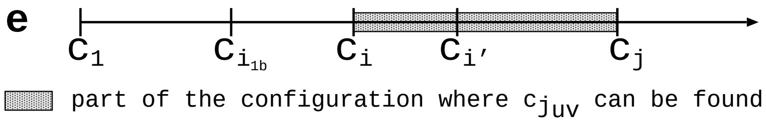

Figure 4.2: According to Q_{1b}, from a configuration $c_{i_{1b}}$, configurations $c_{j_{uv}}$ can be found later than any given configuration c_i.

We show now that, thanks to weak fairness, every execution is in Q_{1b}.

Lemma 9 *Every execution of the* $\mathcal{PA}$*-MP algorithm in the distributed system* $\mathcal{S}$ *is in* Q_{1b}*.*

Proof: Let $e \in \mathcal{E}$ be an execution that is not in Q_{1b}. From Lemma 8, e is in Q_1 and, from a configuration $c_{i_1} \in e_{|\mathcal{C}}$, for every configuration c_j and every link (u, v), the configuration $c_{j_{uv}}$ exists. Now, let us consider configurations c_i, $c_{i'}$ and c_j in $e_{|\mathcal{C}}$ such that $i_1 \leq i \leq i' \leq j$. If $e \notin Q_{1b}$, then configuration $c_{j_{uv}}$ always appears before c_i, even if $c_{i'}$ (and then c_j) is as far as possible from c_i (see Figure 4.3). This means that the values produced by processor u after $c_{j_{uv}}$ were never received, that contradicts Lemma 3.

□

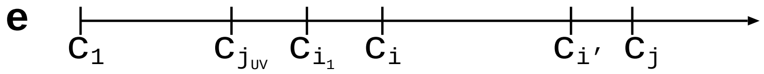

Figure 4.3: If $e \notin Q_{1b}$, the configuration $c_{j_{uv}}$ always appears before c_i.

4.3 Outputs are eventually smaller than legitimate values

Let us begin by defining two predicates P_2 and P_{2b} on configurations. If $P_2(c)$ holds, then, in configuration c, each processor is smaller than all initial values of its ancestors increased by some r-mappings (more precisely, for any processor v and any of its direct-ancestors u, the output of v is smaller than the initial value of u transformed by the r-path-mapping $r_{P_{u,v}}$ of the path $P_{u,v}$ from u to v). If $P_{2b}(c)$ holds, then, in the configuration c, the output of each processor v is smaller (in the sense of $\oplus$) than its legitimate output.

Definition 10 *Let P_2 and P_{2b} be predicates on configurations $c \in \mathcal{C}$:*

$$\begin{array}{lcl} P_2(c) & \equiv & \forall v \in \mathcal{P}, \forall u \in \Gamma_v^-, \forall P_{u,v} \in \mathcal{X}_{u,v}, \quad \mathtt{OUT}_v(c) \preceq_\oplus r_{P_{u,v}}(\mathtt{IN}_u) \\ P_{2b}(c) & \equiv & \forall v \in \mathcal{P}, \quad \mathtt{OUT}_v(c) \preceq_\oplus \widehat{\mathtt{OUT}}_v \end{array}$$

We now define two sets of executions Q_2 and Q_{2b}. If an execution e is in Q_2 (resp Q_{2b}), then there exists a configuration in e from which every configuration satisfies P_2 (resp. P_{2b}).

Definition 11 *Let Q_2 and Q_{2b} be two subsets of $\mathcal{E}$:*

$$\begin{array}{l} \forall e \in Q_2, \quad \exists c_{i_2} \in e_{|\mathcal{C}}, \quad \forall c_j \in e_{|\mathcal{C}} \textit{ with } i_2 \leq j, \quad P_2(c_j) \\ \forall e \in Q_{2b}, \quad \exists c_{i_{2b}} \in e_{|\mathcal{C}}, \quad \forall c_j \in e_{|\mathcal{C}} \textit{ with } i_{2b} \leq j, \quad P_{2b}(c_j) \end{array}$$

We now prove that, first every execution of $\mathcal{E}$ is in Q_2, and then that every execution of $\mathcal{E}$ is in Q_{2b}. This means that, while the processor's outputs can be larger than the legitimate values in the beginning of an execution, each processor eventually produces some outputs that are smaller than or equal to its legitimate value. In other terms, any erroneous values that are larger than legitimate values eventually disappear from $\mathcal{S}$.

Lemma 10 *Every execution of the $\mathcal{PA}$-MP algorithm in the distributed system $\mathcal{S}$ is in Q_2.*

Proof: Let $e \in \mathcal{E}$ be an execution, and let us consider a processor $v_0 \in \mathcal{P}$, and one of its direct-ancestor $v_1 \in \Gamma_{v_0}^{-1}$. By Lemma 7, e is in Q_0. Then, there exists a configuration $c_{i_0} \in e_{|\mathcal{C}}$ such that, for any subsequent configuration $c_{j_{v_0}} \in e_{|\mathcal{C}}$ ($i_0 \leq j_{v_0}$), Predicate $P_{0b}(c_{j_{v_0}})$ is satisfied: $\mathtt{OUT}_{v_0}(c_{j_{v_0}}) \preceq_\oplus r_{v_0}^{v_1}(\mathtt{IN}_{v_0}^{v_1}(c_{j_{v_0}}))$.

Since $e \in Q_1$, the above configuration $c_{j_{v_0}}$ can be chosen after c_{i_1} in e (*i.e.*, $i_0 \leq j_{v_0}$ and $i_1 \leq j_{v_0}$) so that there exists a configuration $c_{j_{v_1 v_0}} \in e_{|\mathcal{C}}$ that appears before $c_{j_{v_0}}$ (*i.e.*, $j_{v_1 v_0} \leq j_{v_0}$) satisfying: $\mathtt{OUT}_{v_1}(c_{j_{v_1 v_0}}) = \mathtt{IN}_{v_0}^{v_1}(c_{j_{v_0}})$. This gives:

$$\mathtt{OUT}_{v_0}(c_{j_{v_0}}) \leq r_{v_0}^{v_1}\left(\mathtt{OUT}_{v_1}(c_{j_{v_1 v_0}})\right) \tag{4.2}$$

Since $e \in Q_{1b}$, it is possible to choose configuration $c_{j_{v_0}}$ in $e_{|\mathcal{C}}$ in order to ensure that $c_{j_{v_1 v_0}}$ appears *after* c_{i_0}. Hence, without loss of generality, we can state $i_0 \leq j_{v_1 v_0}$ and thus $P_{0a}(c_{j_{v_1 v_0}})$ holds. This means that $\mathtt{OUT}_{v_1}(c_{j_{v_1 v_0}}) \preceq_\oplus \mathtt{IN}_{v_1}$, and, from Lemma 2, we have: $r_{v_0}^{v_1}(\mathtt{OUT}_{v_1}(c_{j_{v_1 v_0}})) \preceq_\oplus r_{v_0}^{v_1}(\mathtt{IN}_{v_1})$.

Finally, we obtain the following relation, that remains true for configurations that appear after $c_{j_{v_0}}$:

$$\mathtt{OUT}_{v_0}(c_{j_{v_0}}) \preceq_\oplus r_{v_0}^{v_1}(\mathtt{IN}_{v_1}) \tag{4.3}$$

and this result remains true hereafter.

To iterate the above reasoning from vertex v_1 (instead of v_0) at configuration $c_{j_{v_1 v_0}}$ (instead of $c_{j_{v_0}}$), we must ensure that $c_{j_{v_1 v_0}}$ appears after c_{i_0} (to use Q_0) and after c_{i_1} (to use Q_1). Yet using the fact that $e \in Q_{1b}$, the configuration $c_{j_{v_0}}$ can be chosen as far as necessary in e in order to ensure that the related configuration $c_{j_{v_1 v_0}}$ happens *after* the configurations c_{i_0} and c_{i_1} (see Figure 4.2). Hence, for any path $v_k, \ldots, v_0$, there exists some configurations $c_{j_{v_k v_{k-1}}}, \ldots, c_{j_{v_1, v_0}}, c_{j_{v_0}}$ such that the following relations (obtained from Equations 4.2 and 4.3) remain true for the rest of the execution:

$$\begin{array}{ccccccc}
\mathtt{OUT}_{v_0}(c_{j_{v_0}}) & \preceq_\oplus & r_{v_0}^{v_1}(\mathtt{OUT}_{v_1}(c_{j_{v_1 v_0}})) & \wedge & \mathtt{OUT}_{v_0}(c_{j_{v_0}}) & \preceq_\oplus & r_{v_0}^{v_1}(\mathtt{IN}_{v_1}) \\
\mathtt{OUT}_{v_1}(c_{j_{v_1 v_0}}) & \preceq_\oplus & r_{v_1}^{v_2}(\mathtt{OUT}_{v_1}(c_{j_{v_2 v_1}})) & \wedge & \mathtt{OUT}_{v_1}(c_{j_{v_1 v_0}}) & \preceq_\oplus & r_{v_1}^{v_2}(\mathtt{IN}_{v_2}) \\
\vdots & \preceq_\oplus & \vdots & \wedge & \vdots & \preceq_\oplus & \vdots \\
\mathtt{OUT}_{v_k}(c_{j_{v_k v_{k-1}}}) & \preceq_\oplus & r_{v_k}^{v_{k-1}}(\mathtt{OUT}_{v_0}(c_{j_{v_1 v_0}})) & \wedge & \mathtt{OUT}_{v_k}(c_{j_{v_k v_{k-1}}}) & \preceq_\oplus & r_{v_k}^{v_{k-1}}(\mathtt{IN}_{v_{k-1}})
\end{array} \tag{4.4}$$

Then, for any ancestor v_k of v_0 and any path $P_{v_k v_0} \in \mathcal{X}_{v_k, v_0}$ from v_k to v_0, there exists a configuration $c_{j_{v_0}}$ such that the following remains true in any subsequent configuration: $\mathtt{OUT}_{v_0}(c_{j_{v_0}}) \preceq r_{P_{v_k v_0}}(\mathtt{IN}_{v_k})$. Hence there exists a configuration c_{i_2} reached after all configurations $c_{j_{v_0}}$ (for any processor $v_0 \in \mathcal{P}$) and such that, for any further configuration c_j (*i.e.*, $i_{2b} \leq j$), we have $P_2(c_j)$. This gives the lemma. □

Lemma 11 *Every execution of the $\mathcal{PA}$-MP algorithm in the distributed system $\mathcal{S}$ is in Q_{2b}.*

Proof: Let us consider an execution $e \in \mathcal{E}$. Since $e \in Q_2$, there exists a configuration $c_{i_2} \in e_{|\mathcal{C}}$ such that, for any subsequent configuration $c_j \in e_{|\mathcal{C}}$ (*i.e.*, $i_2 \leq j$), $P_2(c_j)$ holds:

$$\forall v \in \mathcal{P}, \forall u \in \Gamma_v^-, \forall P_{u,v} \in \mathcal{X}_{u,v}, \quad \mathtt{OUT}_v(c_j) \preceq_\oplus r_{P_{u,v}}(\mathtt{IN}_u)$$

Then, we have:

$$\forall v \in \mathcal{P}, \quad \mathtt{OUT}_v(c_j) \preceq_\oplus \bigoplus_{u \in \Gamma_v^-, P_{u,v} \in \mathcal{X}_{u,v}} r_{P_{u,v}}(\mathtt{IN}_u)$$

Since $e \in Q_1$, some of these configurations c_j also satisfy predicate P_{0a}. Without loss of generality, we assume that $P_{0a}(c_j)$ holds: $\mathtt{OUT}_v(c_j) \preceq_\oplus \mathtt{IN}_v$. Hence, we have:

$$\forall v \in \mathcal{P}, \quad \mathtt{OUT}_v(c) \preceq_\oplus \mathtt{IN}_v \oplus \bigoplus_{u \in \Gamma_v^-, P_{u,v} \in \mathcal{X}_{u,v}} r_{P_{u,v}}(\mathtt{IN}_u)$$

This ends the proof, by Definition 4. □

4.4 Legitimate values are eventually reached

Let us begin by defining a predicate on system configurations.

Definition 12 *Let P_3 be a predicate on configurations $c \in \mathcal{C}$:*

$$P_3(c) \quad \equiv \quad \forall v \in \mathcal{P}, \quad \mathtt{OUT}_v(c) = \widehat{\mathtt{OUT}}_v$$

We now define the set of executions Q_3, that corresponds to executions of $\mathcal{E}$ for which every processor eventually reach its legitimate value: all executions of Q_3 reach a configuration c_{i_3} such that, for any subsequent configuration c_j, the outputs of every processor v in c_j are equal to their legitimate values.

Definition 13 *Let $Q_3 \subset \mathcal{E}$ be the set of executions that satisfy:*

$$\forall e \in Q_3, \quad \exists c_{i_3} \in e_{|\mathcal{C}}, \forall c_j \in e_{|\mathcal{C}}, \ \textit{with } i_3 \leq j, \quad P_3(c)$$

We now prove that any execution is in Q_3.

Lemma 12 *Every execution of the $\mathcal{PA}$-MP algorithm in the distributed system $\mathcal{S}$ is in Q_3.*

Proof: Let $e \in \mathcal{E}$ be an execution, and suppose that $e \notin Q_3$. Since $\preceq_\oplus$ defines a total order (Hypothesis 3), we have:

$$\forall c_{i_3} \in e_{|\mathcal{C}}, \quad \exists c_j \in e_{|\mathcal{C}}, \text{ with } i_3 \leq j, \quad \exists v \in \mathcal{P}, \quad \widehat{\mathtt{OUT}}_v \prec_\oplus \mathtt{OUT}_v(c_j) \quad \vee \quad \mathtt{OUT}_v(c_j) \prec_\oplus \widehat{\mathtt{OUT}}_v \tag{4.5}$$

By Lemma 11, e is in Q_{2b} and there exists some configurations c_j that satisfy both $i_3 \leq j$ and $i_{2b} \leq j$, so that $\mathtt{OUT}_v(c_j) \preceq_\oplus \widehat{\mathtt{OUT}}_v$. Hence, Equation 4.5 becomes:

$$\forall c_{i_3} \in e_{|\mathcal{C}}, \quad \exists c_j \in e_{|\mathcal{C}}, \text{ with } i_3 \leq j, \quad \exists v \in \mathcal{P}, \quad \mathtt{OUT}_v(c_j) \prec_\oplus \widehat{\mathtt{OUT}}_v \tag{4.6}$$

By Definition 4 and Lemma 1, we have $\widehat{\mathtt{OUT}}_v \preceq_\oplus \mathtt{IN}_v$. This gives $\mathtt{OUT}_v(c_j) \prec_\oplus \mathtt{IN}_v$. Since $e \in Q_0$, there exists some configurations $c_j \in e_{|\mathcal{C}}$ satisfying both Equation 4.6 and $P_{0c}(c_j)$, that is $i_0 \leq j$. Without loss of generality, we suppose that $P_{0c}(c_j)$ holds: $\exists u \in \Gamma_v^{-1}, \mathtt{OUT}_v(c_j) = r_v^u(\mathtt{IN}_v^u(c_j))$.

As $\mathtt{OUT}_v(c_j) \prec_\oplus \widehat{\mathtt{OUT}}_v$, we have $r_v^u(\mathtt{IN}_v^u(c_j)) \neq e_\oplus$. Since $r_v^u(e_\oplus) = e_\oplus$ (see § 3.3), we have $\mathtt{IN}_v^u(c_j) \neq e_\oplus$. Then, by Definition 2, we have $\mathtt{IN}_v^u(c_j) \prec_\oplus r_v^u(\mathtt{IN}_v^u(c_j))$ and finally $\mathtt{IN}_v^u(c_j) \prec_\oplus \mathtt{OUT}_v(c_j)$. Hence, the following holds: $\exists u \in \Gamma_v^{-1}, \mathtt{IN}_v^u(c_j) \prec_\oplus \mathtt{OUT}_v(c_j)$.

By Lemma 8, $e \in Q_1$, and there exists some configuration c_j that satisfy $i_1 \leq j$ (as well as $i_4 \leq j$, $i_{2b} \leq j$ and $i_0 \leq j$) and for which configuration $c_{j_{uv}}$ exists in e and verifies $\mathtt{OUT}_u(c_{j_{uv}}) = \mathtt{IN}_v^u(c_j)$. Then $\mathtt{OUT}_u(c_{j_{uv}}) \prec_\oplus \mathtt{OUT}_v(c_j) \prec_\oplus \widehat{\mathtt{OUT}}_v$. This means that at least one of the direct-ancestors u of v verifies $\mathtt{OUT}_u(c_{j_{uv}}) \prec_\oplus \widehat{\mathtt{OUT}}_u \vee \widehat{\mathtt{OUT}}_u \prec_\oplus \mathtt{OUT}_u(c_{j_{uv}})$ (indeed, if all ancestors of v reached and hold their legitimate value, then v would reach its legitimate value too). Hence, Equation 4.6 becomes:

$$\begin{gathered}\forall c_{i_3} \in e_{|\mathcal{C}}, \quad \exists c_j \in e_{|\mathcal{C}}, \text{ with } i_3 \leq j, \quad \exists u, v \in \mathcal{P} \text{ with } u \in \Gamma_v^{-1}, \quad \exists c_{j_{uv}} \in e_{|\mathcal{C}}, \text{ with } j_{u,v} \leq j \\ \left(\mathtt{OUT}_u(c_{j_{uv}}) \prec_\oplus \widehat{\mathtt{OUT}}_u \quad \vee \quad \mathtt{OUT}_u(c_{j_{uv}}) \prec_\oplus \widehat{\mathtt{OUT}}_u\right) \quad \wedge \quad \mathtt{OUT}_u(c_{j_{uv}}) \prec_\oplus \mathtt{OUT}_v(c_j) \prec_\oplus \widehat{\mathtt{OUT}}_v\end{gathered} \tag{4.7}$$

To iterate the above argument from processor u instead of v, and from configuration $c_{j_{uv}}$ instead of c_j, we argue that $i_0 \le j_{uv}$, $i_1 \le j_{uv}$, and $i_{2b} \le j_{uv}$. By Lemma 9, e is in Q_{1b}. This means that configurations c_{i_3} in the above equation can be chosen so that every configurations $c_{j_{uv}}$ appear *after* configurations c_{i_0}, c_{i_1} and $c_{i_{2b}}$ (see Figure 4.2). This allows to re-use the above reasoning with configuration $c_{j_{uv}}$ instead of c_j.

By iterating the above arguments, and since the network is finite, we exhibit a cycle of nodes and a set of configurations $c_{j_0}, c_{j_1} \ldots$ appearing after c_{i_4} in e such that, for a node w in the cycle, we have:

$$\mathtt{OUT}_w(c_{j_0}) \prec_\oplus \mathtt{OUT}_w(c_{j_1}) \prec_\oplus \ldots \prec_\oplus \widehat{\mathtt{OUT}}_w \tag{4.8}$$

Using the fact that $e \in Q_{1b}$, this can be found after any configuration c_{i_4} in the execution e. This means that, regardless of configuration c_{i_4}, there exists subsequent configurations $c_{j_0}, \ldots c_{j_1}$, such that $\widehat{\mathtt{OUT}}_w$ increases strictly without reaching its legitimate value. We then exhibit a strictly increasing sequence of values of $\mathbb{S}$ that never reach $\widehat{\mathtt{OUT}}_w$. This is impossible if $\mathbb{S}$ is finite. If $\mathbb{S}$ is infinite, then Lemma 6 gives $\widehat{\mathtt{OUT}}_w \prec_\oplus e_\oplus$. The sequence of values is then upper bounded, that contradicts Hypothesis 4. Hence, $e \in Q_3$. $\Box$

4.5 Complexity

In the convergence part of the proof, we only assumed that computations were maximal, and that message loss, duplication and desequencing could occur. In order to provide an upper bound on the stabilization time for our algorithm, we assume strong synchrony between processors and a reliable communication medium between nodes. Note that these assumptions are used for complexity results only, since our algorithm was proved correct even in the case of asynchronous unfair computations with link intermittent failures. In the following, D denotes the network diameter.

In order to give an upper bound on the space and time requirements, we assume that the set $\mathbb{S}$ is finite, and that $|\mathbb{S}|$ denotes its number of elements. Note that this assumption is used for complexity results only, since our algorithm was proved to be correct even in the case when $\mathbb{S}$ is infinite.

The space complexity result is immediately given by the assumptions made when writing Algorithm $\mathcal{PA}$-MP.

Lemma 13 (Space Complexity) *Each processor $v \in \mathbb{S}$ holds $(\delta v + 1) \times \log_2(|\mathbb{S}|)$ bits.*

Proof: Each processor v has δv local variables that hold the value of the last message sent by the corresponding direct ancestor, and one register used to communicate with its direct descendants. Each of these local variables may hold a value in a finite set $\mathbb{S}$, then need $\log_2(|\mathbb{S}|)$ bits. Note that the constant stored in ROM is not taken into account in this result. $\Box$

Lemma 14 (Time Complexity) *Assuming a synchronous system $\mathcal{S}$, the stabilization time is $O(D + |\mathbb{S}|)$.*

Proof: We define ϕ as the function that returns the index of a given element of $\mathbb{S}$. This index always exists since $\mathbb{S}$ is ordered by a total order relation. The signature of ϕ is as follows:

$$\begin{array}{rccc} \phi : & \mathbb{S} & \to & \mathbb{N} \\ & s & \mapsto & \phi(s) \end{array} \qquad \text{and} \qquad s_1 \prec_{\oplus} s_2 \quad \Rightarrow \quad \phi(s_1) < \phi(s_2)$$

After $O(D)$ steps, every node in the network has received values from all of their ancestors. If those values were badly initialized, then the received values are also possibly badly valued.

For each node u, we consider the difference between the index of its final value (since the algorithm converges to a legitimate configuration where $\mathtt{OUT}_u = \widehat{\mathtt{OUT}_u}$) and the index of the smallest received value which is badly initialized. The biggest possible difference is $M - m$, where M is the maximum index value of $\mathbb{S}$ and m the minimum index value of $\mathbb{S}$. This difference is called d and is $O(|\mathbb{S}|)$.

For each node u, we also consider the smallest and the greatest (in the sense of increasing) r-path mapping from u to u. Let l be the length of the smallest such r-path mapping. It increases a value index by at least l. The greatest such r-path mapping increases a value index by at most d, and is of length at most d.

In the worst case, there exists a node that has an incorrect input value indexed with m, a correct input value indexed with M, so it has to wait that the incorrect value index is increased by $M-m$ before the incorrect value effect is canceled. Each l times units at least, this incorrect value index is increased by l. Again, in the worst case, if $\lfloor \frac{d}{l} \rfloor < \frac{d}{l}$, another incorrect value may still be lower than the correct value, and the greatest cycle may be followed, inducing an extra d time delay. Overall, after the first $O(D)$ times units, $\left(\lfloor \frac{d}{l} \rfloor \times l\right) + d = O(d)$ time units are needed. □

Chapter 5

Concluding remarks

Summary. We presented a parameterized distributed algorithm applicable to any directed graph topology. This algorithm tolerates transient faults that corrupt the processors and communication links memory (it is self-stabilizing) as well as intermittent faults (fair loss, reorder, finite duplication of messages) on communication media.

Applications. The function parameter of our algorithm can be instantiated [19, 20] to produce distributed algorithms for both fundamental and high level applications. We quickly sketch two possible applications of the generic algorithm. First, to solve the shortest path problem with r-operators, it is sufficient to consider $\mathbb{N} \cup \{+\infty\}$ as $\mathbb{S}$, $+\infty$ as $e_\oplus$, min as $\oplus$, and $x \mapsto x + c_{u,v}$ as r_u^v. 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 from 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 $x \mapsto x \times \tau_u^v$ as r_u^v (where τ_u^v is the reliability rate of the edge between u and v, with $0 < \tau_u^v < 1$) our parameterized algorithm ensures that a best transmitter tree is maintained despite transient failures (in a self-stabilizing way).

Future works. We are now investigating the possibility of maintaining invariants (*e.g.* a route toward a destination if the r-operator used calculates a shortest path toward this destination) while the inputs of the system are changing (*i.e.* the r-mappings are evolving during the execution of the system), and still preserve the self-stabilizing and unreliable communications tolerance of our algorithm.

Bibliography

[1] Y Afek and G M Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7:27–34, 1993.

[2] Y Afek, S Kutten, and M Yung. Memory-efficient self-stabilization on general networks. In *WDAG90 Distributed Algorithms 4th International Workshop Proceedings, Springer-Verlag LNCS:486*, pages 15–28, 1990.

[3] E Anagnostou and V Hadzilacos. Tolerating transient and permanent failures. In *Proceedings of WDAG'93, LNCS 725*, pages 174–188, 1993.

[4] A Arora, P Attie, M Evangelist, and M Gouda. Convergence of iteration systems. *Distributed Computing*, 7:43–53, 1993.

[5] B Awerbuch, B Patt-Shamir, and G Varghese. Self-stabilization by local checking and correction. In IEEE, editor, *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 268–277, San Juan, Porto Rico, October 1991. IEEE Computer Society Press.

[6] B Awerbuch, B Patt-Shamir, G Varghese, and S Dolev. Self-stabilization by local checking and global reset. In *Proceedings of WDAG'94, LNCS 857*, volume 857, pages 326–??, 1994.

[7] F Baccelli, G Cohen, G Olsder, and J-P Quadrat. *Synchronization and Linearity, an algebra for discrete event systems*. Wiley, Chichester, UK, 1992.

[8] A Basu, B Charron-Bost, and S Toueg. Simulating reliable links in the presence of process crashes. In *Proceedings of the Tenth International Workshop on Distributed Algorithms (WDAG'96), LNCS 1151*, 1996.

[9] J Beauquier and S Kekkonen-Moneta. Fault tolerance and self-stabilization: impossibility results and solutions using self-stabilizing failure detectors. *International Journal of Systems Science*, 28(11):1177–1187, november 1997.

[10] A Bui, A K Datta, F Petit, and V Villain. State-optimal snap-stabilizing PIF in tree networks. In *Proceedings of the Fourth Workshop on Self-stabilizing Systems*, pages 78–85, 1999.

[11] J A Cobb and M G Gouda. Stabilization of routing in directed networks. In *Proceedings of the Fifth Internationa Workshop on Self-stabilizing Systems (WSS'01), Lisbon, Portugal*, pages 51–66, 2001.

[12] S Delaët and S Tixeuil. Tolerating transient and intermittent failures. *Journal of Parallel and Distributed Computing*, 62(5):961–981, 2002.

[13] E W Dijkstra. Self-stabilization in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[14] S. Dolev. *Self-stabilization.* The MIT Press, 2000.

[15] S Dolev, MG Gouda, and M Schneider. Memory requirements for silent stabilization. *Acta Informatica*, 36(6):447–462, 1999.

[16] S Dolev and T Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 3(4), 1997.

[17] S Dolev, A Israeli, and S Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[18] 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*, 1998.

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

[20] B Ducourthial and S Tixeuil. Self-stabilization with path algebra. *Theoretical Computer Science*, 293(1):219–236, 2003.

[21] A. S. Gopal and K. J. Perry. Unifying self-stabilization and fault-tolerance. In *Proceedings of PODC'93*, pages 195–206, 1993.

[22] J-H Hoepman, M. Papatriantafilou, and P Tsigas. Self-stabilization of wait free shared memory objects. *Journal of Parallel and Distributed Computing*, 62(5):818–842, 2002.

[23] N A Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[24] T Masuzawa. A fault-tolerant and self-stabilizing protocol for the topology problem. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 1.1–1.15, 1995.

[25] G Tel. *Introduction to Distributed Algorithms.* Cambridge university press, 1994.

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