RAPPORT de RECHERCHE LRI n° 1202, février 1999

Self-stabilization with Path Algebra

Bertrand Ducourthial Sébastien Tixeuil

Laboratoire de Recherche en Informatique (bât. 490),
Université Paris-Sud, 91405 Orsay Cedex, France
email: {ducourth,tixeuil}@lri.fr

Abstract

Self-stabilizing protocols can resist transient failures and guarantee system recovery in a finite time. We highlight the connexion between the formalism of self-stabilizing distributed systems and the formalism of generalised path algebra and asynchronous iterations with delay. We use the later to prove that a local condition on locally executed algorithm (being a strictly idempotent r-operator) ensures self-stabilization of the global system. As a result, a parametrized distributed algorithm applicable to any directed graph topology is proposed, and the function parameter of our algorithm is instantiated to produce distributed algorithms for both fundamental and high level applications. Due to fault resilience properties of our algorithm, the resulting protocols are self-stabilizing at no additional cost.

Keywords : Distributed Algorithms, Self-stabilization, Max-plus Algebra, r-operators.

Résumé

Les protocoles auto-stabilisants peuvent résister aux défaillances transitoires en garantissant un retour à la normale du système en un temps fini. Nous mettons l'accent sur les relations existantes entres les systèmes répartis auto-stabilisants d'une part et l'alèbre de chemins d'autre part. Nous utilisons le formalisme des itérations asynchrones avec délai afin de prouver qu'une condition locale sur l'algorithme exécuté par un processeur du système (être un r-opérateur strictement idempotent) assure l'auto-stabilisation du système tout entier. En utilisant ce résultat, nous proposons un algorithme paramétré applicable à un graphe orienté de topologie quelconque. L'opérateur paramétrant notre algorithme est utilisé de manière à produire des protocoles résolvant à la fois des problèmes fondamentaux et de haut niveau. Étant données les propriétés de tolérance aux défaillances de notre algorithme, les protocoles résultants de ces instanciations sont auto-stabilisants sans surcoût.

Mots Clés : Algorithmes Répartis, Auto-stabilisation, Algèbre Max-plus, r-opérateurs.

Chapter 1

Introduction

Robustness is one of the most important requirements of modern distributed systems. Various types of faults are likely to occur at various parts of the system. These systems go through the transient faults because they are exposed to constant change of their environment.

1.1 Self-stabilization

One of the most inclusive approaches to fault tolerance in distributed systems is *self-stabilization* [12, 25]. Introduced by Dijkstra in [12], this technique guarantees that, regardless of the initial state, the system will eventually converge to the intended behavior or the set of *legitimate* states. Since most self-stabilizing fault-tolerant protocols are non-terminating, if the distributed system is subject to transient faults corrupting the internal node state but not its behavior, once faults cease, the protocols themselves guarantee to recover in a finite time to a safe state without the need of human intervention. This also means that the complicated task of initializing distributed systems is no longer needed, since self-stabilizing protocols regain correct behavior regardless of the initial state. Furthermore, note that in practice, the context in which we may apply self-stabilizing algorithms is fairly broad since the program code can be stored in a stable storage at each node so that it is always possible to reload the program after faults cease or after every fault detection.

1.2 Related Work

Silent tasks [13] are tasks where the communication between the processors is fixed from some point of the execution. In addition to simplicity implied by the silence property, silent algorithms may use fewer resources in terms of communication operations and communication bandwidth allocation. In our model, registers are used for communication between processors. Then a system solving a silent task has the property that the contents of the communication registers is not changed after some point in the execution. When the algorithm checks that a register needs to be changed before performing a write operation, all write operations may be eliminated when the silent system has reached a legitimate configuration. Examples of silent tasks include leader election, spanning tree construction or single source shortest path algorithms. Note that many tasks fundamental to distributed systems are inherently non silent. Such tasks include mutual exclusion or token passing, where the contents of communication registers have to change infinitely often in every possible execution of the system.

Historically, research in self-stabilization over general networks has mostly covered undirected networks where bidirectional communication is feasible (the Update protocol of [14], or the algorithms presented in [2, 15]). Bidirectional communication is usually heavily used in bidirectional self-stabilizing systems to compare one node state with those of its neighbors and check for consistency. The self-stabilizing algorithms that are built upon the paradigm of local checking (see [6, 7]) use this scheme.

The lack of bidirectional communication was overcome in recent papers using several techniques. Strong connectivity (which is a weaker requirement than bidirectionality) was assumed to build a virtual well known topology on which the self-stabilizing algorithm may be run (a tree in [1], a ring in [4]). As many self-stabilizing algorithms exists for rings ([12]) or trees ([3]) in the literature, these constructions may be used to reuse existing algorithms in general networks.

The restriction of having either bidirectional communication media or strongly connected unidirectional networks are reasonable when the task to be solved is dynamic and the system is asynchronous: *e.g.* for traversal algorithms, a token has to be able to pass through every node infinitely often. However, there exists several silent tasks for which global communication is not required. For example, the single source shortest path task only requires that a directed path exists from a node called the *source* to any other node, but not the converse. [5] used the formalism of Iteration Systems to give sufficient conditions for convergence of systems solving related tasks. Silent tasks have been solved in a self-stabilizing way on directed graphs that are not strongly connected in [10], but the underlying network was assumed having no cycle (DAG). The absence of cycles permits to avoid cases where corrupted data moves forever in the system, preventing it from stabilizing.

1.3 Our Contribution

In this paper, we concentrate on solving silent tasks in a self-stabilizing way on a truly general network, where no hypothesis are made about the strong connectivity or the presence of cycles. As in [5], our solution is by giving a condition on the distributed algorithm. However, in [5], 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.

In this paper, we extend results presented in [16], emphasizing the modeling of distributed computations with asynchronous multiplications of max-plus matrices. The contribution is twofold. First we provide a way of modeling a class of silent self-stabilizing distributed algorithm through the formalism of max-plus algebra, using a matrix representation of the system. Then we extend a result presented in [18] using this formalism. To this purpose, 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-stabilizing provided that these tasks can be expressed through local calculus operations called r-operators. 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. Our system performs in the composite atomicity model, under the general fully distributed demon (see [29]). In this model, at each step of the system computation, the global scheduler of the system (*aka* the demon) may choose a subset of the available processors to run their algorithm, but each processor may compute its value at a different speed.

1.4 Outline of the paper

The rest of the paper is organized as follows. In Chapter 2, we give some definitions pertinent to the protocols and proofs. The self-stabilizing parameterized protocol is presented in Chapter 3 along with the r-operators used for local computations and the matrix modeling based on path-algebra and asynchronous iterations. The correctness reasoning for the parameterized protocol is given in Chapter 4. Application to fundamental problems in distributed computing area are presented in Chapter 5. We discuss the extension of our ideas and make some concluding remarks in Chapter 6.

Chapter 2

Self-stabilizing Distributed Systems

In Section 2.1, we define the concept of a distributed system, basically a graph whose vertices model processors and arcs model the communication media allowing them to exchange information. In this context, self-stabilizing distributed systems are introduced in Section 2.2 and defined as systems that eventually match a given specification in spite of transient failures that may corrupt all volatile variables.

2.1 Distributed Systems

2.1.1 Underlying Graph

A distributed system $\mathcal{S}$ is a collection of N processors linked with communication media allowing them to exchange information. 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[1] pairs of vertices of V ($v_1, v_2 \in V$). Each vertex u in V represents a processor P_u of the system $\mathcal{S}$. Each edge (u, v) in E, represents a communication link from P_u to P_v in $\mathcal{S}$. We give now some graph definitions.

The *in-degree* of a vertex v of G denoted by $\delta^-(v)$ 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 numbered from 1 to $\delta^-(v)$.

A *directed path* from a vertex v_0 to a vertex v_k in a digraph $G(V, E)$ is a list of consecutive edges of E, $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$. The length of this path is k. If each v_i is unique in the path, the path is *elementary.* A *cycle* is a directed path where $v_0 = v_k$. A digraph without any cycle is called a *directed acyclic graph (DAG).*

The *distance* between two vertices u, v of a digraph G, denoted by $\mathrm{d}_G(u, v)$, is the minimum of the lengths of all directed paths from u to v. The *diameter* of a digraph G, denoted by $\mathrm{Diam}(G)$, is the maximum of the distances between all couples of vertices in G. The *strongly connected component* of a vertex v in a digraph $G(V, E)$ is the set of all vertices w

[1] $(v_1, v_2) \neq (v_2, v_1)$.

of V such that there exists a directed path from v to w and a directed path from w to v. G is *strongly connected* if it has exactly one strongly connected component.

The *direct descendants* of a vertex v of a digraph $G(V,E)$ are all the vertices w of G such that the edge (v,w) is in E. Their set is denoted by $\Gamma_G^{+1}(v)$. Similarly, the *direct ancestors* of a vertex v of G are all the vertices u of G such that the edge (u,v) is in E. Their set is denoted by $\Gamma_G^{-1}(v)$. We denote by $\overline{\Gamma_G^{-1}(v)}$ the set $\Gamma_G^{-1}(v) \cup \{v\}$. The *ancestors* of v are all the vertices u such that there exists a path from u to v. Their set is denoted by $\Gamma_G^{-}(v)$.

2.1.2 Communications

A communication from processor P_u to processor P_v is only feasible if the vertex u is a direct ancestor of the vertex v in G (*i.e.* (u,v) is an edge of G). Such a communication is performed as follows. Processor P_u writes the datum to be sent to P_v into a dedicated shared register. Then P_v is able to read the datum into this register and to use it. A processor may only write into its own shared register and can only read shared registers owned by its direct ancestor processors or itself.

Hypothesis 1 *Any communication in the distributed system $\mathcal{S}$ is done through shared registers.*

In addition to those shared registers, processors may maintain local variables when executing their code. Such local variables are private to the processor and can not be accessed by any of its neighbors.

2.1.3 Processors

A processor is a deterministic sequential machine that runs a single process. The *state* of a processor is defined by the values of its local variables. The state of a link (u,v) of E is defined by the value of the associated shared register. A processor *action* (or step) consists of a `read` action, then an internal computation followed by a `write` action. Internal action of processors are not significant to its neighbors because they have no access to the variables that are manipulated by those actions. `read` and `write` actions are the only way for two processors to communicate.

2.1.4 Configuration and Executions

Classical definitions for configurations and executions of distributed systems can be found in [28]. A *configuration* of a distributed system $\mathcal{S}$ is an instance of the states of its processors and links. The set of configurations of $\mathcal{S}$ is denoted as $\mathcal{C}$. Processor actions change the global system configuration. An *execution* e (also called a *computation*) is a sequence of configurations $c_1, c_2, \ldots$ such that for $i = 1, 2, \ldots$, the configuration c_{i+1} is reached from c_i by a single step of at least one processor. Configuration c_1 is the *initial configuration* of execution e.

The set of executions in the distributed system $\mathcal{S}$ starting with a particular initial configuration $c_1 \in \mathcal{C}$ is denoted by $\mathcal{E}_{c_1}$. Every execution $e \in \mathcal{E}_{c_1}$ is of the form $c_1, c_2, \ldots$. The set of execution in the system $\mathcal{S}$ whose initial configurations are all elements of $\mathcal{C}_1 \subset \mathcal{C}$ is denoted as $\mathcal{E}_{\mathcal{C}_1}$. The set $\mathcal{E} = \mathcal{E}_{\mathcal{C}}$ contains all possible executions of system $\mathcal{S}$.

All executions considered in this paper are assumed to be *maximal* meaning that the sequence is either infinite, or it is finite and no action is enabled in the final configuration. An algorithm is *silent* if for each possible execution, either of the two following conditions is verified : (i) the execution is finite or (ii) the execution is infinite and there exists a configuration c_t such that any subsequent execution (in $\mathcal{E}_{c_t}$) contains only c_t configurations.

To model the non-deterministic behavior of a distributed system, we assume processor activity is managed by a global *scheduler*. To ensure correctness of the system, we regard the scheduler as an adversary. Each processor that is chosen by the adversary executes exactly one atomic step. The adversary may be more or less powerful depending on (i) the freedom it has in choosing the activated processors and (ii) the grain of the atomicity. More freedom and finer atomicity grain gives the adversary more power. We refer to the most common types of adversaries used in the literature and more specifically in [15, 22, 29]:

1. The *synchronous demon* activates simultaneously all of the system's processors. Then, all processors read their input registers, perform local computations and write their output register.

2. The *distributed demon* can activate simultaneously any subset of the system's processors. When such a subset is activated, all processors in the subset read their input registers, perform local computations and write their output register.

3. The *fully distributed demon* can activate simultaneously any subset of the system's processor. When such a subset is activated, all processors in the subset read their input registers, perform local computations, but may delay writing their output register after the following demon activation occured. However, a processor may not be chosen again by the demon until it has wrote its output registers.

4. The *read/write demon* activates a single processor at a time. When a processor is activated, it may either read exactly one input register or (not both) write its output register. Contrary to the preceeding demons that use composite atomicity (any processor read *all* its input registers atomically, and write *all* its output registers atomically), the Read/Write demon uses register atomicity.

The strongest adversary is the Read/Write demon, while the weakest adversary is the Synchronous demon. As an immediate corollary, if a distributed system works correctly under the scheduling of the strongest demon, it will also perform correctly under a weaker adversary.

Hypothesis 2 *We assume an intermediate adversary, the fully distributed demon.*

2.2 Self-stabilization

2.2.1 Specifications

A *specification* is a predicate on executions that are admissible for a distributed system. A system *matches its specification* if all its possible executions match the specification. If we consider only *static* problems (*i.e.*, problems whose solutions consist of computing some global result), the specification can be given in terms of a set of configurations. Every execution matching the specification would be a sequence of such configurations. The set of configurations that matches the specification of static problems is called the set of *legitimate* configurations (denoted as $\mathcal{L}$), while the remainder $\mathcal{C} \setminus \mathcal{L}$ denotes the set of *illegitimate* configurations.

2.2.2 Self-stabilization

We need to introduce the concept of an attractor to define self-stabilization. Intuitively, an attractor is a set of configurations of the system $\mathcal{S}$ that "attracts" another set of configurations of $\mathcal{S}$ for any execution in $\mathcal{E}$. In addition, if the attractor is closed, then any subsequent execution of the algorithm remains in the same set of configurations.

Definition 1 (Closed Attractor) *Let $\mathcal{C}_a$ and $\mathcal{C}_b$ be subsets of $\mathcal{C}$. $\mathcal{C}_a$ is an attractor for $\mathcal{C}_b$ if and only if for any initial configuration c_1 in $\mathcal{C}_b$, for any execution e in $\mathcal{E}_{c_1}$, ($e = c_1, c_2, \ldots$), there exists $i \geq 1$ such that for any $j \geq i$, $c_j \in \mathcal{C}_a$.*

Definition 2 (Self-stabilization) *A system $\mathcal{S}$ is called self-stabilizing if and only if there exists a non-empty subset $\mathcal{L} \subset \mathcal{C}$ of legitimate configurations such that $\mathcal{L}$ is a closed attractor for $\mathcal{C}$.*

In the usual (*i.e.* non-stabilizing) distributed systems, possible executions can be restricted by allowing the system to start only from some well-defined *initial* configurations. On the other hand, in stabilizing systems, problems can not be solved using this convenience, since all possible system configurations are admissible initial configurations.

Chapter 3

Self-stabilizing Global Computations with Path Algebra

The purpose of this section is twofold. First we recall previous results concerning relations between distributed systems and operator-based algorithms for the one hand, and between function-weighted graphs and path algebra for the other hand. Then we extend previous results to model silent distributed systems using the matrix representation used in asynchronous iterations. The rest of the section is organized as follows : Section 3.1 indicates when silent distributed systems can be seen as algorithms that perform local operator-based computations; Section 3.2 recall previous results about r-operators, which generalize the concept of Tel's infimum functions; Section 3.3 recall previous results about generalized path algebra and Section 3.4 shows how silent distributed systems can be modeled using asynchronous iterations.

3.1 Silent Distributed Systems as Operator-based Algorithms

In this section, we define the concept of guarded rule and describe distributed systems that perform a global calculus using a parametric algorithm that simply gets input data from its incoming neighbors, computes a local function $\mathcal{F}_v$ and finally makes the result of this function available to its outgoing neighbors. Since silent distributed algorithms have their communication fixed from some point in each execution, we can consider this point as the global result computed by the algorithm.

3.1.1 Programming Notation

The program for each protocol consists of a sequence of actions: *action*, ... ,*action*. Each *action* has the form: $<$ *guard* $>\longrightarrow$ *statement*. A *guard* is a boolean expression over the local variables of a processor and the communication registers of its immediate ancestors. A *statement* is allowed to update the communication register of the processor only. Any action whose guard is *true* is *enabled.* A processor with one or more enabled actions is said to be *privileged* and may make a *move* executing the statement corresponding to the chosen enabled guard.

Hypothesis 3 *We assume a* fair *adversary, i.e. in any infinite execution, if a processor is infinitely often privilegied, then it is chosen by the adversary infinitely often.*

3.1.2 Parametric Algorithm

Each processor P_v has two local constants stored in Read Only Memory: the *initial datum*, $\mathsf{ROM}[v]$, and the set of its direct ancestors $\Gamma_G^{-1}(v)$. To store the result of the local computation, a single register is used at P_v: $\mathsf{RES}[v]$, the *outgoing variable.* Such a register is used for the communications between P_v and all its direct descendants (*one-to-many communication scheme*). Each processor P_v also has access to the communication register $\mathsf{RES}[w]$ of any of its direct ancestors $w \in \Gamma_G^{-1}(v)$, the *incoming variables.*

In addition to the above, the protocol maintains on each processor P_v a function $\mathcal{F}_v$ which is defined as:

$$\begin{array}{lll} \mathcal{F}_v : & \mathtt{A}^{\delta^-(\mathtt{v})+1} & \to \mathtt{A} \\ & \mathsf{ROM}[v], \mathsf{RES}[u_1], \ldots, \mathsf{RES}[u_{\delta^-(v)}] & \mapsto \mathcal{F}(\mathsf{ROM}[v], \mathsf{RES}[u_1], \ldots, \mathsf{RES}[u_{\delta^-(v)}]) \end{array}$$

where nodes u_1 through $u_{\delta^-(v)}$ are the direct ancestors of v. Then each processors v runs as follows : v performs a local computation using function $\mathcal{F}_v$, its initial datum $\mathsf{ROM}[v]$ and the incoming data $\mathsf{RES}[u_1], \ldots \mathsf{RES}[u_{\delta^-(v)}]$, and then stores the result into its outgoing variable $\mathsf{RES}[v]$. This guarded rule is parametrized at each node v by $\mathcal{F}_v$ and is shown in Algorithm 1.

Algorithm 1 (The $\mathcal{PA}$ algorithm) *Algorithm $\mathcal{PA}$ at node v consists in one rule R parametrized by $\mathcal{F}_v$:*

$R|\mathcal{F}_v$: $\langle \mathsf{true} \rangle$

$\longrightarrow$ $\mathit{RES}[v] \leftarrow \mathcal{F}_v(\mathit{ROM}[v], \mathit{RES}_{u\in\Gamma_G^{-1}(v)}[u])$

where $\mathit{RES}_{u\in\Gamma_G^{-1}(v)}[u]$ stands for the sequence $\mathit{RES}[u_1], \ldots \mathit{RES}[u_{\delta^-(v)}]$ with u_1 through $u_{\delta^-(v)}$ being the direct ancestors of v.

Note: Having each node v disposing of the $\Gamma_G^{-1}(v)$ is only convenient when writing the algorithm. In an actual implementation, this set can be efficiently replaced by the list of v in-port hardware addresses.

3.2 r-operators

In this section, we define r-operators as an extension of Tel's infimum functions. We will use a slightly different version of the r-operators than defined in [16], that is sufficient for our purpose. In Section 3.2.4, we recall results from previous work presented in [18] that show the relationship between r-operators and self-stabilizing distributed systems.

3.2.1 Infimum Functions

Our distributed algorithm is parametrized at each node v in system $\mathcal{S}$ with function $\mathcal{F}_v$. The $\mathcal{F}_v$ function computes a result from v direct ancestors' values. In Chapter 5, we only define functions $\mathcal{F}_v$ for each $v \in \mathcal{S}$ to describe the whole system. In the following, we investigate sufficient conditions on the $\mathcal{F}_v$ functions so that the system is self-stabilizing for a given specification.

In [27], Tel proves infimum computations terminate when the $\mathcal{F}_v$ function is an infimum over the set of inputs. An *infimum* (hereby called a *s-operator*) $\oplus$ over the 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: $x \preceq_\oplus y$ if and only if $x \oplus y = x$. Moreover, [27] assumes that there exists a greatest element on $\mathbb{S}$, denoted by $\top$, and verifying $x \preceq_s \top$ for every $x \in \mathbb{S}$. If necessary, this element can be added to $\mathbb{S}$.

Hence, the $(\mathbb{S}, \oplus)$ structure is an *abelian idempotent semigroup*[1]. Using $\oplus$ as $\mathcal{F}_v$ in our parametrized algorithm yields a silent distributed system, yet [18] proved that the resulting protocol is not self-stabilizing.

3.2.2 Binary r-operators

Starting from Tel results, [17] introduced a distorted algebra – the r-algebra – by generalizing properties of the abelian idempotent semigroup with a mapping r. An *r-operator* is a dissymmetric s-operator, that we usually denote by $\triangleleft$:

Definition 3 (r-operator) *The operator $\triangleleft$ is an r-operator on $\mathbb{S}$ if there exists a bijective mapping r from $\mathbb{S}$ to $\mathbb{S}$ such that $\triangleleft$ verifies the following properties :* **a.** *r-associativity: $(x \triangleleft y) \triangleleft \mathsf{r}(z) = x \triangleleft (y \triangleleft z)$;* **b.** *$r$-commutativity: $\mathsf{r}(x) \triangleleft y = \mathsf{r}(y) \triangleleft x$;* **c.** *$r$-idempotency: $\mathsf{r}(x) \triangleleft x = \mathsf{r}(x)$.*

For example, the operator $\mathsf{minc}(x, y) = \mathsf{min}(x, y + 1)$ is an idempotent r-operator on $\mathbb{Z} \cup \{+\infty\}$. The mapping r (which is $x \mapsto x+1$ for minc) is called r-mapping of the r-operator. This mapping is unique and when it is equal to the identity ($x \mapsto x$), the corresponding r-operator is an s-operator. The r-operators have many applications in parallel and distributed computing (see [16, 17] for further details). We recall some of their algebraic properties. For any r-operator, there exists an s-operator $\oplus$ such that for any x and y in $\mathbb{S}$, $x \oplus y = x \triangleleft \mathsf{r}^{-1}(y)$. The identity element $\top$ of $\oplus$ is the right identity element of $\triangleleft$. Moreover the r-mapping of any r-operator is an isomorphism of $(\mathbb{S}, \triangleleft)$ and $(\mathbb{S}, \oplus)$. There are as many r-operators on $\mathbb{S}$ as couples of s-operators $\oplus$ and isomorphisms r of $(\mathbb{S}, \oplus)$. When the r-operator $\triangleleft$ verifies $x \triangleleft x = x$ for any x in $\mathbb{S}$, it is *idempotent*. Constructing an idempotent r-operator $\triangleleft$ from an s-operator $\oplus$ and an isomorphism r of $(\mathbb{S}, \oplus)$ is done by assuming that $\oplus$ and r verify, for any x in $\mathbb{S}$, $x \preceq_\oplus \mathsf{r}(x)$. When for each x in $\mathbb{S}$, $x \prec_\oplus \mathsf{r}(x)$ (*i.e.* $x \preceq_s y$ and $x \neq y$),

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

the r-operator $\lhd$ is *strictly idempotent.* For some proofs, we suppose that the r-operator $\lhd$ verifies: $\forall y, z \in \mathbb{S}, (\forall x \in \mathbb{S}, \quad x \lhd y = x \lhd z) \quad \Leftrightarrow \quad (y = z)$. In practice, this hypothesis is verified by many operators.

3.2.3 n-ary r-operators

Binary r-operators can be extended to accept an arbitrary number of arguments. A mapping $\lhd$ from $\mathbb{S}^n$ into $\mathbb{S}$ is an *n-ary r-operator* if there exists an s-operator $\oplus$ on $\mathbb{S}$ and $n-1$ isomorphisms $\mathsf{r}_1, \ldots, \mathsf{r}_{n-1}$ of $(\mathbb{S}, \oplus)$ such that

$$\lhd(x_0, \ldots, x_{n-1}) = x_0 \oplus \mathsf{r}_1(x_1) \oplus \cdots \oplus \mathsf{r}_{n-1}(x_{n-1})$$

for any $x_0, \ldots, x_{n-1}$ in $\mathbb{S}$. In other words, an n-ary r-operator consists in $n-1$ binary r-operators based on the same s-operator. If all of these binary r-operators are (strictly) idempotent, the resulting n-ary r-operator is (strictly) idempotent.

Note: Although more general r-operators have been defined in [16] using r-mappings of multiple variables, we do not consider such operators in this paper. Moreover, we assume that r-mappings are not bijective. As a consequence, some of the aforementioned algebraic properties do not hold in this context. Hopefully, the self-stabilizing proofs presented in Chapter 4 only requires that the s-operator $\oplus$ defines an ordering relation and that the r-mappings are strictly expansive ($x \prec_{\oplus} \mathsf{r}(x)$).

Hypothesis 4 (r-operator) *An n-ary r-operator $\lhd$ is defined on $\mathbb{S}$ from an s-operator $\oplus$ on $\mathbb{S}$, and $n-1$ endomorphisms $\mathsf{r}_1, \ldots, \mathsf{r}_{n-1}$ of $(\mathbb{S}, \oplus)$ as follows: for all $x_0, \ldots, x_{n-1} \in \mathbb{S}$, $\lhd(x_0, \ldots, x_{n-1}) = x_0 \oplus \mathsf{r}_1(x_1) \oplus \cdots \oplus \mathsf{r}_{n-1}(x_{n-1})$.*

In addition, we suppose that the following properties on the r-mappings hold:

Hypothesis 5 (Strict Idempotency) *For any r-mapping r_i, for any $i \in \{1, \ldots, |E|\}$ and for any $x \in \mathbb{S}$, $x \prec_{\oplus} \mathsf{r}_i(x)$.*

Hypothesis 6 ($\mathsf{r}_\perp$) *Let $\mathsf{r}_\perp$ be the endomorphism built using all of the r-mappings $\mathsf{r}_1, \ldots, \mathsf{r}_{|E|}$ used in the distributed system $\mathcal{S}$ as follows: $\forall x \in \mathbb{S}, \mathsf{r}_\perp(x) = \bigoplus_{i \in \{1, \ldots, |E|\}} \mathsf{r}_\perp(x)$ Then $\lim_{k \to +\infty} \mathsf{r}_\perp^k(x) = e_{\oplus}$.*

3.2.4 Self-stabilization with r-operators

When our distributed algorithm is instantiated with an r-operator $\lhd$ as function $\mathcal{F}_v$, each node $v \in \mathcal{S}$ performs :

$$\begin{array}{lr} \mathsf{RES}[v] = \mathsf{ROM}[v] \oplus \mathsf{r}(\mathsf{RES}[u_1]) \oplus \cdots \oplus \mathsf{r}(\mathsf{RES}[u_{\delta^-(v)}]) & \textit{binary r-operator} \\ \mathsf{RES}[v] = \mathsf{ROM}[v] \oplus \mathsf{r}_1\,(\mathsf{RES}[u_1]) \oplus \cdots \oplus \mathsf{r}_{\delta^-(v)}(\mathsf{RES}[u_{\delta^-(v)}]) & \textit{n-ary r-operator} \end{array}$$

where $u_1, \ldots, u_{\delta^-(v)}$ denote node v direct ancestors, $\mathsf{r}_1, \ldots, \mathsf{r}_{\delta^-(v)}$ denote the corresponding r-mappings, and $\oplus$ denotes the s-operator $\lhd$ is based upon.

[16] showed that when the r-operator is idempotent, the algorithm is silent, and [18] proved that when it is strictly idempotent and $\oplus$ induces a total order on $\mathbb{S}$, it is also self-stabilizing for the following specifications :

$$\mathsf{RES}[v] = \bigoplus \left\{ \mathsf{r}^{\mathrm{d}_G(u,v)}\left(\mathsf{RES}[u]\right), u \in \overline{\Gamma_G^-(v)} \right\} \qquad \textit{binary}$$
$$\mathsf{RES}[v] = \bigoplus \left\{ \mathsf{r}_P\left(\mathsf{RES}[u]\right), u \in \overline{\Gamma_G^-(v)}, P \text{ elementary path from } u \text{ to } v \right\} \qquad \textit{n-ary}$$

where r_P is the composition of the r-mappings corresponding to the edges of the path P. The proof of stabilization in [18] has been established for a read/write demon, assuming the s-operator $\oplus$ defined a total order relation. In this paper, we extend this result and prove – using path algebra – that even if the s-operator $\triangleleft$ is based upon defines a partial order relation on $\mathbb{S}$, our parametric algorithm instantiated with $\triangleleft$ is self-stabilizing. This extension leads to new applications, that we present in Chapter 5. However, we restrict executions by assuming a fully distributed demon.

3.3 Path algebra

Definitions related to max-plus algebra can be found in [8, 20, 21].

3.3.1 Semi-algebra

A *semiring* $(\mathbb{S}, \oplus, \otimes)$ is defined by the four following conditions : ***a.*** $(\mathbb{S}, \oplus)$ is an abelian semigroup which identity element is $e_\oplus$, ***b.*** $\otimes$ is associative and admits an identity element $e_\otimes$, ***c.*** $\otimes$ is distributive over $\oplus$ and ***d.*** $e_\oplus$ is absorbing for $\otimes$. The structure is idempotent when $\oplus$ is idempotent.

A *semimodule* $(\mathbb{M}, \oplus, \cdot)$ over the semiring $(\mathbb{S}, \oplus, \otimes)$ is a set of matrices $\mathbf{A}$ such that the following conditions hold : ***a.*** $(\mathbb{M}, \oplus)$ is an abelian semigroup which $\oplus$ is defined by $(\mathbf{A} \oplus \mathbf{B})[i][j] = \mathbf{A}[i][j] \oplus \mathbf{B}[i][j]$ and which identity element is denoted as $\mathbf{E}_\oplus$, ***b.*** $\cdot$ is an external composition law defined by $(\lambda \cdot \mathbf{A})[i][j] = \lambda \cdot \mathbf{A}[i][j]$ with $\lambda \in \mathbb{S}$ and $\mathbf{A} \in \mathbb{M}$, ***c.*** $\oplus$ and $\cdot$ are distributive and $\cdot$ verifies $\alpha(\beta\mathbf{A}) = \beta(\alpha\mathbf{A})$ for all α and β in $\mathbb{S}$ and ***d.*** $e_\oplus\mathbf{A} = \mathbf{E}_\oplus$ and $e_\otimes\mathbf{A} = \mathbf{A}$.

Let $\otimes$ define a matrix multiplication: $(\mathbf{A} \otimes \mathbf{B})[i][j] = \bigoplus_{k=1}^{k=N} \mathbf{A}[i][k] \otimes \mathbf{B}[k][j]$. When endowed with a third law such that $(\mathbb{M}, \oplus, \otimes)$ is a semiring, $(\mathbb{M}, \oplus, \otimes, \cdot)$ is a *semi-algebra*. The identity element of $\otimes$ is denoted by $\mathbf{E}_\otimes$. When the semifield $\mathbb{S}$ is idempotent, the semialgebra is also idempotent.

3.3.2 Graph Interpretation

There exists a duality between matrix operations and graph theory. The part of Max-plus algebra that studies graph theory is known in the literature as *path algebra* (see for example [11]).

The *precedence graph* $G[\mathbf{A}]$ associated to an $N \times N$ matrix $\mathbf{A}$ of the semi-algebra $(\mathbb{M}, \oplus, \otimes, \cdot)$ is a directed graph with N vertices numbered from 1 to N, such that there exists an arc from j to i if and only if $\mathbf{A}[i][j] \neq e_\oplus$. When different from $e_\oplus$, $\mathbf{A}[i][j]$ is called *weight* of the arc (j, i). The weight $w(P)$ of a directed path $P = (v_0, v_1) \dots (v_{k-1}, v_k)$ in $G[\mathbf{A}]$ is equal to the product by $\otimes$ of the weights of the edges of P: $w(P) = \mathbf{A}[v_0][v_1] \otimes \cdots \otimes \mathbf{A}[v_{k-1}][v_k]$. We use the following standard notations: $\mathbf{A}^{(k)} = \mathbf{E}_\otimes \oplus \mathbf{A} \oplus \cdots \oplus \mathbf{A}^k$ and $\mathbf{A}^* = \lim_{k \to +\infty} \mathbf{A}^{(k)}$. A circuit C is *absorbing* if its weight $w(C)$ verifies $e_\otimes \oplus w(C) \neq e_\otimes$.

In [20], Gondran showed that if $\mathbf{A}$ is an $N \times N$ matrix in a semi-algebra, the following properties hold:

1. $(\mathbf{A}^k)[i][j]$ is the sum (in sense of $\oplus$) of the weights of the paths from j to i having exactly k edges,
2. $(\mathbf{A}^{(k)})[i][j]$ is the sum of the weights of the paths from j to i having at most k edges.
3. if $G[\mathbf{A}]$ has no absorbing circuit, then there exists an integer $p < N$ such that $\mathbf{A}^* = \mathbf{A}^{(p)}$.

3.3.3 Generalized Path Algebra

In [21, 24], an extension was provided, allowing to label edges with a function instead of a scalar. Let $(\mathbb{S}, \oplus)$ be an abelian idempotent semi-group, with $\mathsf{e}_\oplus$ as its neutral element, and let $\mathbb{H}$ be the set of the endomorphisms over $(\mathbb{S}, \oplus)$. Let the $\oplus$ law of $\mathbb{S}$ be extended to $\mathbb{H}$ by $(\mathsf{h}_1 \oplus \mathsf{h}_2)(x) = \mathsf{h}_1(x) \oplus \mathsf{h}_2(x)$ and let $\mathsf{e}_\oplus \in \mathbb{H}$ (defined by $\mathsf{e}_\oplus(x) = e_\oplus$ for all $x \in \mathbb{S}$) be its neutral element. Let $\otimes$ be the composition law of two morphisms of $\mathbb{H}$: $(\mathsf{h}_1 \otimes \mathsf{h}_2)(x) = \mathsf{h}_2(\mathsf{h}_1(x))$ and let $\mathsf{e}_\otimes : x \mapsto x$ be its neutral element. Then $(\mathbb{H}, \oplus, \otimes)$ is an idempotent semiring.

In particular, given these notations, if for each edge (i, j) in the graph $G[\mathbf{A}]$, its label $\mathsf{h}_{ij} \in \mathbb{H}$ verifies $\mathsf{h}_\otimes \oplus \mathsf{h}_{ij} = \mathsf{h}_\otimes$, the graph has no absorbing circuit, and $\mathbf{A}^*$ exists.

3.4 Silent Distributed Systems as Asynchronous Iterations

3.4.1 System Configurations as Vectors

The distributed systems we consider have processors communicating through shared registers. Since we consider executions under the synchronous, distributed and fully distributed demons, which use composite atomicity, processors read all their incoming registers in a single atomic step. Since processors hold a single output register, a given configuration of the system is modelled by a vector of register values (one entry per processor). During an execution, we have a sequence of such vectors, where $\mathtt{X}_n$ denotes the n^{th} vector describing the system configuration. The successive evolutions of the system during each executions are properly described through the corresponding evolutions of the configuration vector.

We now review the synchronous, distributed and fully distributed demons, and their relationship with asynchronous iterations.

3.4.2 The Synchronous Demon

As detailed in Section 3.2, when parametrized by an n-ary r-operator $\lhd$ based on an s-operator $\oplus$, our distributed algorithm leads to the following local computations:

$$\mathsf{RES}[v] = \mathsf{ROM}[v] \oplus \mathsf{r}_1 \left(\mathsf{RES}[u_1]\right) \oplus \cdots \oplus \mathsf{r}_{\delta^-(v)}(\mathsf{RES}[u_{\delta^-(v)}]) \tag{3.1}$$

Under the synchronous demon, all nodes perform this local computation simultaneously. Thus, the system computations can be written using a matrix notation. Let $(\mathbb{S}, \oplus)$ be the idempotent abelian semigroup corresponding to the s-operator associated to the n-ary r-operator $\lhd$. Let $(\mathbb{H}, \oplus, \otimes)$ be the idempotent semiring of the endomorphisms of $\mathbb{S}$, let $\mathsf{e}_\oplus$ and $\mathsf{e}_\otimes$ be its zero (neutral element for $\oplus$) and unity (neutral element for $\otimes$), respectively. Let $(\mathbb{M}, \oplus, \otimes, \cdot)$ be the idempotent semi-algebra over $(\mathbb{H}, \oplus, \otimes)$ of the $N \times N$ matrices (see Section 3.3). Now consider the $N \times N$ precedence matrix $\mathbf{A} \in \mathbb{M}$, associated to system $\mathcal{S}$ (composed of N nodes): $\mathbf{A}[i][j] = \mathsf{e}_\oplus$ if the edge (i, j) does not exist, and $\mathbf{A}[i][j] = \mathsf{r}_{ij}$ if the edge (i, j) exists and r_{ij} is its r-mapping.

Let $\mathtt{X}_n$ be the vector composed of the values stored in the outgoing registers of each of the N nodes of system $\mathcal{S}$ at step n. Let $\mathtt{B}$ be the vector composed of the values stored in the ROMs of those nodes in the same order. Then, we have:

$$\mathtt{X}_{n+1} = \mathbf{A} \otimes \mathtt{X}_n \quad \oplus \quad \mathtt{B} \tag{3.2}$$

In the following, F denotes the vector operator: $\mathtt{X} \mapsto \mathtt{X} = \mathbf{A} \otimes \mathtt{X} \quad \oplus \quad \mathtt{B}$.

3.4.3 The Distributed Demon

The distributed demon does not necessarily activate all processors at the same time. Thus, Equation 3.2 does not hold in this case. Still it is possible to consider global computations steps, but we must take into account that all processors may not participate in building a new value at each step. Let $\mathcal{J}_n$ be the set of processors activated at step n by the distributed demon. From equation 3.2, we obtain:

$$\mathtt{X}_{n+1}[i] = \begin{cases} \mathtt{X}_n[i] & \text{if } i \notin \mathcal{J}_n \\ \mathsf{F}\left(\mathtt{X}_n\right) = \mathbf{A} \otimes \mathtt{X}_n \quad \oplus \quad \mathtt{B} & \text{if } i \in \mathcal{J}_n \end{cases} \tag{3.3}$$

Informally, Equation 3.3 reads as only selected processors (those of $\mathcal{J}_n$) compute a new value using the last produced result of their direct ancestors. Such equations are known in the literature as *asynchronous iterations* (see [9, 31]).

3.4.4 The Fully Distributed Demon

Under control of the fully distributed demon, activated processors do not necessaryly write their output registers within the same round. Thus Equation 3.3 does not hold in this context. Hence, a processor may compute its output value using its last read input values, while those input values have changed in between. This enforces the asynchronism between processors because they do not have to perform similar operations (computing their local algorithm) within the same time (as it is the case under the synchronous or distributed demon).

We introduce a delay in Equation 3.3 to model this time lap between `read` and `write` actions. At step n, supposing $i \in \mathcal{J}_n$, processor P_i uses the data of its direct ancestor P_j produced at step $D_n[j]$ (instead of $n-1$). Delaying leads to the following *asynchronous iteration with delay* (see [23, 31]):

$$\mathtt{X}_{n+1}[i] = \begin{cases} \mathtt{X}_n[i] & \text{if } i \notin \mathcal{J}_n \\ \mathsf{F}\left((\mathtt{X}_{D_n[1]}[1], \dots, \mathtt{X}_{D_n[N]}[N])^{\mathrm{t}}\right) = & \\ \qquad \mathbf{A} \otimes (\mathtt{X}_{D_n[1]}[1], \dots, \mathtt{X}_{D_n[N]}[N])^{\mathrm{t}} \oplus \mathtt{B} & \text{if } i \in \mathcal{J}_n \end{cases} \tag{3.4}$$

3.4.5 Conditions for Convergence of Asynchronous Iterations

Asynchronous iterations have been extensively studied for optimization purpose on parallel computers (see [9, 19, 23, 30, 31, 32]). Under particular conditions, asynchronous iterations (Equations 3.3 and 3.4) converge to the same result as *synchronous iterations* (Equation 3.2), while reducing data dependency.

In [31], Üresin and Dubois give several sufficient conditions ensuring the convergence of asynchronous iterations. They also point out that their work can be applied to path algebra. Since the vector operator F is defined on a Cartesian product of (possibly infinite) sets $\mathbb{S}$ ordered by the relation $\preceq_\oplus$ (which is extended to $\mathbb{S}^N$), any asynchronous iteration converges for any initial guess $\mathtt{X}_0 \in \mathbb{S}^N$ if the following conditions are verified (see Proposition 3 page 599 in [31] concerning finite or infinite sets): ***a.*** F is closed on $\mathbb{S}^N$; ***b.*** the synchronous iterations converge, and $\mathtt{X}_{n+1} \preceq \mathtt{X}_n$ for each $n \in \mathbb{N}$; ***c.*** F is monotonous on $\mathbb{S}^N$, that is, for all $\mathtt{X}$ and $\mathtt{Y}$ in $\mathbb{S}^N$, $\mathtt{X} \preceq_\oplus \mathtt{Y}$ implies $\mathsf{F}(\mathtt{X}) \preceq_\oplus \mathsf{F}(\mathtt{Y})$.

When these conditions are fulfilled, F has a fixed-point in $\mathbb{S}^N$ such that every asynchronous iteration corresponding to Equation 3.3 converges.

In order to prove that our algorithm is self-stabilizing, we need to prove that for any underlying topology, for any initial vector configuration (the initial values in the RES registers), and for any successive choices of the fully distributed demon, its convergence is assured. Note that Equation 3.4 is more general than Equation 3.3 ($D_n[i] = n-1$ for any index i and any step n). In turn, Equation 3.3 is more general than Equation 3.2 ($\mathcal{J}_n = \{1, \dots, N\}$ for any step n). This order on equations mimics the total order on demons we mentioned in Section 2.1.4. Consequently, the proof of stabilization given in the following section is only established for the fully distributed demon.

Chapter 4

Proving Self-stabilization using Path Algebra

In this section we prove the self-stabilization property of the parametric algorithm instanciated by a strictly idempotent r-operator, using path algebra and asynchronous iterations, under the fully distributed demon.

Theorem 1 *When the parametric algorithm A is instanciated with a strictly idempotent n-ary r-operator, it is self-stabilizing on any topology.*

Proof. An n-ary r-operator $\lhd$ defined on the set $\mathbb{S}$ is built from an s-operator $\oplus$ and $n-1$ r-mappings r_i which are endomorphisms of $\mathbb{S}$ (see Hypothesis 4):

$$\lhd(x_0, \dots, x_{n-1}) = x_0 \oplus \mathsf{r}_1(x_1) \oplus \cdots \oplus \mathsf{r}_{n-1}(x_{n-1})$$

Such an operator leads to the following local computations on nodes of the distributed system (see § 3.2.4):

$$\mathsf{RES}[v] = \mathsf{ROM}[v] \oplus \mathsf{r}_1\left(\mathsf{RES}[u_1]\right) \oplus \cdots \oplus \mathsf{r}_{\delta^-(v)}(\mathsf{RES}[u_{\delta^-(v)}])$$

Let $(\mathbb{H}, \oplus, \otimes)$ be the idempotent semiring of the endomorphisms over the semigroup $(\mathbb{S}, \oplus)$, and $(\mathbb{M}, \oplus, \otimes, \cdot)$ be the semi-algebra of the $N \times N$ matrices composed of elements of $\mathbb{H}$ (see § 3.3.1). Let $\mathbf{A} \in \mathbb{M}$ be the $N \times N$ precedence matrix associated to the distributed system, where each entry $\mathbf{A}[i][j]$ is the r-mapping corresponding to the edge (j, i) if it exists, and $\mathsf{e}_\oplus$ else (see § 3.3.2 and § 3.3.3).

Assuming the fully distributed demon, and using asynchronous iterations (see § 3.4.4), the global computations performed by the whole distributed system can be modelled as follows:

$$\mathtt{X}_{n+1}[i] = \begin{cases} \mathtt{X}_n[i] & \text{if } i \notin \mathcal{J}_n \\ \mathsf{F}\left((\mathtt{X}_{D_n[1]}[1], \dots, \mathtt{X}_{D_n[N]}[N])^{\mathrm{t}}\right) = & \\ \qquad \mathbf{A} \otimes (\mathtt{X}_{D_n[1]}[1], \dots, \mathtt{X}_{D_n[N]}[N])^{\mathrm{t}} \oplus \mathtt{B} & \text{if } i \in \mathcal{J}_n \end{cases} \tag{4.1}$$

Proving the self-stabilization of Algorithm 1 on any topology is equivalent to proving the convergence of these asynchronous iterations with delay for *any initial vector* $\mathtt{V}_0$ and any precedence matrix $\mathbf{A}$. The vector operator F is defined on $\mathbb{S}^N$, which is a cartesian product. We then have to verify that the Üresin and Dubois following conditions hold (see § 3.4.5):

1. the vector operator F is closed on $\mathbb{S}^N$;

2. F is monotonous on $\mathbb{S}^N$;

3. the synchronous iterations converge.

We successively prove that these conditions are verified.

1. From definition of F, the first condition is straightforward.

2. Let $\mathtt{X}_1$ and $\mathtt{X}_2$ be two vectors of $\mathbb{S}^N$ such that $\mathtt{X}_1 \preceq_{\oplus} \mathtt{X}_2$. We then have $\mathtt{X}_1 \oplus \mathtt{X}_2 = \mathtt{X}_1$ and $\mathtt{X}_1 \otimes \mathbf{A} :: \oplus :: \mathtt{X}_2 \otimes \mathbf{A} = \mathtt{X}_1 \otimes \mathbf{A}$ and $(\mathtt{X}_1 \otimes \mathbf{A} :: \oplus :: \mathtt{B}) :: \oplus :: (\mathtt{X}_2 \otimes \mathbf{A} :: \oplus :: \mathtt{B}) = \mathtt{X}_1 \otimes \mathbf{A} :: \oplus :: \mathtt{B}$. The second condition is thus fullfilled.

3. There remains to prove that synchronous iterations $\mathtt{X}_{k+1} = \mathbf{A} \otimes \mathtt{X}_k :: \oplus :: \mathtt{B}$ converge for *any initial guess* $\mathtt{X}_0$.

We have
$$\begin{array}{rcll}
\mathtt{X}_1 &=& \mathbf{A} \otimes \mathtt{X}_0 :: \oplus :: \mathtt{B} & \\
\mathtt{X}_2 &=& \mathbf{A} \otimes \mathtt{X}_1 :: \oplus :: \mathtt{B} &= \mathbf{A}^2 \otimes \mathtt{X}_0 :: \oplus :: \mathbf{A} \otimes \mathtt{B} :: \oplus :: \mathtt{B} \\
& \dots & &
\end{array}$$
which leads to
$$\begin{aligned}
\mathtt{X}_k &= \mathbf{A}^k \otimes \mathtt{X}_0 :: \oplus :: \mathbf{A}^{k-1} \otimes \mathtt{B} :: \oplus :: \cdots :: \oplus :: \mathbf{A} \otimes \mathtt{B} :: \oplus :: \mathtt{B} \\
&= \mathbf{A}^k \otimes \mathtt{X}_0 :: \oplus :: \left(\mathbf{A}^{k-1} \oplus \cdots \oplus \mathbf{A} \oplus \mathbf{E}_{\otimes}\right) \otimes \mathtt{B} \\
&= \mathbf{A}^k \otimes \mathtt{X}_0 :: \oplus :: \mathbf{A}^{(k-1)} \otimes \mathtt{B} \quad \text{(see § 3.3.2)}
\end{aligned} \tag{4.2}$$

Let $\mathsf{r}_{\perp}$ be the largest element of $\mathbb{H}$ which is smaller than all the component of the matrix $\mathbb{A}$ (in sense of $\preceq_{\oplus}$): $\mathsf{r}_{\perp} \stackrel{\text{def.}}{=} \bigoplus_{1\leq:i,j:\leq n} \mathbf{A}[i][j]$ Let $\mathbf{R}_{\perp}$ be the matrix of $\mathbb{M}$ defined by $\mathbf{R}_{\perp}[i][j] = \mathsf{r}_{\perp}$, for all indexes i and j. We have: $\mathbf{R}_{\perp} \preceq_s \mathbf{A}$ which leads to $\mathbf{R}_{\perp}^k \preceq_{\oplus} \mathbf{A}^k$. Thanks to hypothesis 6, we have $\lim_{k\to+\infty} \mathsf{r}_{\perp}^k = \mathsf{e}_{\oplus}$. Hence $\lim_{k\to+\infty} \mathbf{R}_{\perp}^k = \mathbf{E}_{\oplus}$. Thus $\lim_{k\to+\infty} \mathbf{A}^k = \mathbf{E}_{\oplus}$.

Since the r-operator is strictly idempotent (Hypothesis 5), $\mathsf{h}_{\otimes} \oplus \mathsf{r}_i = \mathsf{h}_{\otimes}$ for all $i \in \{1, \dots, |E|\}$ and there is no absorbing circuit in the network. Thus $\lim_{k\to+\infty} \mathbf{A}^{(k-1)} = \mathbf{A}^*$ (see § 3.3.2). We then have from Equation 4.2:

$$\begin{array}{l}
\forall : \mathtt{X}_0 \in \mathbb{S}^n, \quad \exists k_0 \in \mathbb{N} \quad \text{such that,} \quad \text{for } k \geq k_0, \\
\mathtt{X}_k \quad = \quad \mathbf{A}^k \otimes \mathtt{X}_0 :: \oplus :: \mathbf{A}^{(k-1)} \otimes \mathtt{B} \quad = \quad \mathbf{A}^* \otimes \mathtt{B}
\end{array}$$

and the synchronous iteration $\mathtt{X}_{k+1} = \mathbf{A} \otimes \mathtt{X}_k :: \oplus :: \mathtt{B}$ converge for *any initial guess* $\mathtt{X}_0$. The third condition is then verified.

¿From results of [31], the theorem is proved. □

Chapter 5

Applications

In this section, we briefly give some examples of r-operators designed to solve particular problems. In some applications given in this section, the initial data are only used to differentiate a particular set of nodes from the others, leading to a semi-uniform algorithm: all nodes execute the same overall code, but based on some hardwired values, some internal functions may not have the same behaviour. On current networks, it is common to have unique identifiers for all nodes (through the hardware network device) so those assumptions do not clash with self-stabilization assumptions.

In [18], several applications that use r-operators based on s-operators that define total order relations are given. These results remain valid here because we have a weaker requirement on the s-operator (defining a partial ordering relation is sufficient). Among the operators presented in [18], **minc** is defined as $\mathbf{minc}(x,\ y)$=$\mathbf{min}(x,\ y+1)$, and is used to solve distance computation, shortest dipath spanning tree and forest. Others problems such as single and multiple source shortest paths, depth-first-search tree are solved using operators $\mathbf{minc}_w$ and **lexicat**, respectively.

In the following, we give other examples of r-operators that solve different problems: the best reliable path from some transmitters is a variant of the shortest paths problem that uses a different "metric" (the s-operator it is based on induces a total ordering relation), while the ordered ancestor list is an application that was not possible in the framework of [18] (since the s-operator it is based on induces only a partial ordering relation).

5.1 Best reliable path from some transmitters

In a telecommunication network where some terminals must chose their "best" transmitter, distance is not always the relevant criterium. When the failure rate of any inter-nodes connection is evaluable and keeps more or less constant, it could be interesting to know the transmitter form where there exists a less failure rate path, and to know the path itself.

Assume that τ_v^i is the failure rate on the i^h incoming edge of the node v and $0 < \tau_v^i \leq 1$. We then denote by $\overline{\tau_v^i}$ the reliable rate of this edge: $\overline{\tau_v^i} = 1 - \tau_v^i$. The reliable rate of a path is the product of the reliable rate of all its edges. We define the n-ary r-operator $\mathsf{Maxtimes}_{\overline{\tau}}$

on $\mathbb{S} = [0,1] \cap \mathbb{R}$ by ($n = \delta^{-(v)} + 1$):

$$\mathsf{Maxtimes}_{\overline{\mathcal{T}}}(x_0, \ldots, x_{\delta^-(v)}) = \mathsf{Max}\left(x_0, x_1 \times \overline{\tau_v^1}, \ldots, x_{\delta^-(v)} \times \overline{\tau_v^{\delta^-(v)}}\right)$$

The n-ary r-operator $\mathsf{Maxtimes}_{\overline{\mathcal{T}}}$ is based on the s-operator Max that defines on $\mathbb{S}$ a total ordering relation $\preceq_{\mathsf{Max}}$ which is, in fact, the usual order $\geq$. Moreover, each r-mapping $r_v^i(x) = x \times \overline{\tau_v^i}$ verifies $x \prec_{\mathsf{Max}} r_v^i(x)$ (which means that $x > r(x)$). Thus the n-ary r-operator is strictly idempotent. In addition, Hypothesis 6 holds.

Initializing all the transmitters to 1 and all the terminals to 0 ($\mathsf{ROM}[v] = 1$ if v is a transmitter, $\mathsf{ROM}[v] = 0$ else), the best reliable path problem is solved after stabilization of Algorithm 1. Indeed, the best path is maintained by the knowledge, on each node, of one incoming variable containing the smallest datum of all those in the incoming variables.

If there are several reachable tranmitters, then each node will find one of the best for it, and this transmitter could send data to it with the less failure rate.

5.2 Ordered Ancestor List

In networks subject to transient failures or dynamic topology changes, up to date topology related information is crucial. We provide in this section an operator that maintains an ordered list of ancestors for any node v in the network. Assuming that nodes have unique indexes over the network, let $\mathbb{S}$ be the set of lists of sets of indexes. For example, if a, b, c and d are processor indexes, then the list $(\{d\}, \{b\}, \{a, c\})$ is an element of $\mathbb{S}$. We define the list concatenation operator $\cdot$ and the binary equivalence relation $\mathcal{R}$ as $l_1 \mathcal{R} l_2 \equiv l_2 = l_1 \cdot (\emptyset)$ and consider all operators are defined on the set $\mathbb{S}/\mathcal{R}$.

Indexes being noted as letters, we consider operator $\oplus$ on $\mathbb{S}/\mathcal{R}$ which merges term to term elements of lists of $\mathbb{S}/\mathcal{R}$, while removing elements already appearing in the resulting list. For example:

$$\begin{array}{rcl} (\{d\}, \{b\}, \{a, c\}) \oplus (\{c\}, \{a, e\}, \{b\}) & = & (\{d, c\}, \{b, a, e\}, \{\cancel{a}, \cancel{c}, \cancel{b}\}) \\ & = & (\{d, c\}, \{b, a, e\}) \end{array}$$

Operator $\oplus$ can be shown being an s-operator. Now consider the mapping r from $\mathbb{S}/\mathcal{R}$ to $\mathbb{S}/\mathcal{R}$ which maps list $l = \{S_1, \ldots, S_k\}$ to list $\mathsf{r}(l) = \{\emptyset, S_1, \ldots, S_k\}$. For example:

$$\mathsf{r}\left((\{d\}, \{b\}, \{a, c\}) \right) = \left(\{\emptyset, \{d\}, \{b\}, \{a, c\}\} \right)$$

Let Ant be a binary r-operator defined with the s-operator $\oplus$ and the r-mapping as $\mathsf{Ant}(l_1, l_2) = l_1 \oplus \mathsf{r}(l_2)$. Mapping r is an endomorphism of $\mathbb{S}/\mathcal{R}$. Operator Ant is strictly idempotent because (i) $(S_1, \ldots, S_k) \oplus (\emptyset, S_1, \ldots, S_k) = (S_1, \ldots, S_k)$ and (ii) $(S_1, \ldots, S_k) \neq (\emptyset, S_1, \ldots, S_k)$ leads to (iii) $(S_1, \ldots, S_k) \preceq_{\oplus} \mathsf{r}(S_1, \ldots, S_k)$. From our theorem and Hypothesis 6, we can conclude that any execution satifies the specification. When the ROM of each node v contains list $(\{v\})$, the result in $\mathsf{RES}[v]$ after stabilization is:

$$\mathsf{RES}[v] \quad = \quad \bigoplus \left\{ r^{\mathrm{d}_G(u,v)}(\{u\}), u \in \Gamma_G^-(v) \right\}$$

This expression is a complete representation of v's ancestors ordered through distance to v. From such information, the diameter of the graph may be infered as the size of the longest list on the network.

Chapter 6

Conclusion

Describing distributed systems using r-operators is convenient due to the local expression of computations occuring in the global system. When actually implementing scalable distributed algorithms, each processor local code has no knowledge of the global topology or configuration. In addition, restrictions imposed by r-operators on the local calculus performed at each node are weak. In practice, finding endomorphisms that are strictly idempodent and such that Hypothesis 6 holds is easy and enables a large set of applications to be written. Chapter 5 showed that both fundamental and high level tasks in distributed computing could be implemented using r-operators.

Max-plus algebra and the matrix representation take the orthogonal approach, using global entities to model the whole system configuration. This permits proving very fine conditions on both circuits and associated operators in order that the distributed system converges to a desirable configuration. While having a condition on each local operator (being a strictly idempotent r-operator) is stronger than having a condition on each circuit and associated operators (no absorbing circuit), it is easier using our approach (i) to write generic proofs that work on any topology directed graphs, and (ii) to verify that a given algorithm satisfies the conditions.

The main contribution of this paper is the connection between asynchronous iterations and their matrix representation for the one hand, distributed systems and their different scheduling policies for the other hand. This connection was done thanks to max-plus algebra, which permitted to enhance significantly previous results in distributed computing, such as [5, 18]. Also worth noticing is the huge reduction of the stabilization proof compared to [18], while remaining generic both from the topological and scheduling point of view.

Among further developments, the authors would like to point out two current research directions:

1. finding new r-operators that solve currently open problems in self-stabilizing distributed systems,

2. finding a similar approach to model other communications schemes widely used in distributed systems (e.g. message passing).

Bibliography

[1] Y. Afek and A. Bremler. *Self-stabilizing Unidirectionnal Network Algorithms by Power Supply.* Chicago Journal of Theoritical Computer Science, Vol. 1998, Art. 3, The MIT Press, December 7, 1998.

[2] Y. Afek, S. Kutten and M. Yung. *Memory-efficient self-stabilization on general networks.* In Workshop on Distributed Algorithms, pp. 15-28, 1990.

[3] L. O. Alima, J. Beauquier, A. K. Datta and S. Tixeuil. *Self-stabilization with Global Rooted Synchronizers.* Proceedings of the 18$^{\text{th}}$ International Conference on Distributed Computing Systems (ICDCS'98), 1998.

[4] D. Alstein, J.-H. Hoepman, B. E. Olivier and P. I. A. Van Der Put. *Self-stabilizing mutual exclusion on directed graphs.* In Computer Science in the Netherlands (Utrech, The Netherlands, 1994), E. Backer (Ed.), Stichting MathematischCentrum, Amsterdam, pp. 42-53.

[5] A. Arora, P. Attie, M. Evangelist and M.G. Gouda. *Convergence of Iteration Systems.* Distributed Computing, Vol. 7, pp. 43–53, 1993.

[6] B. Awerbuch, B. Patt-Shamir and G. Varghese. *Self-stabilization by Local Checking and Correction.* In IEEE Annual Symposium on Foundations of Computer Science, pp. 268–277, Los Alamitos, CA, October 1991.

[7] B. Awerbuch, B. Patt-Shamir, G. Varghese and S. Dolev. *Self-stabilization by Local Checking and Global Reset.* In International Workshop on Distributed Algorithms, pp. 326–339, Berlin, 1994.

[8] F. Baccelli, G. Cohen, G. Olsder and J.-P. Quadrat. *Synchronisation and Linearity, an algebra for discrete event systems.* Series in Probability and Mathematical Statistics. Wiley, Chichester, UK, 1992.

[9] G. M. Baudet. *Asynchronous Iterative Methods for Multiprocessors.* Journal of the ACM, Vol. 25, No. 2, pp. 226–244, April 1978.

[10] B. Bourgon, S. K. Das, A. K. Datta, and V. Natarajan. *A Self-Stabilizing Shortest Path Algorithm in a DAG.* Fourteenth IEEE International Phoenix Conference on Computers and Communications, Scottsdale, Arizona, pp. 341-345, March 29-30, 1995.

[11] P. de la Torre and C. Kruskal. *Fast parallel algorithms for all-sources lexicographic search and path-algebra problems.* *Journal of the Algorithms* Vol. 19 pp. 1–24, 1995.

[12] E. W. Dijkstra. *Self-Stabilizing Systems in Spite of Distributed Control.* Communications of the ACM Vol. 17, No. 11 pp. 643-644, 1974.

[13] S. Dolev, G. Gouda, and M. Schneider. *Memory Requirements for Silent Stabilization.* Proc. of the 15th Annual ACM Symp. on Principles of Distributed Computing, pp. 27-34, 1996.

[14] S. Dolev and T. Herman. *SuperStabilizing Protocols for Dynamic Distributed Systems.* Chicago Journal of Theoritical Computer Science, Vol. 1997, Art. 4, The MIT Press, December 19, 1997.

[15] S. Dolev, A. Israeli and S. Moran. *Self Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity.* Distributed Computing, Vol. 7, pp. 3-16, 1993.

[16] B. Ducourthial. *Les Réseaux Associatifs, un modèle de programmation à parallélisme de données irréguliers, à primitives de calcul asynchrones.* Ph.D. Thesis, Université de Paris Sud, France, January 1999.

[17] B. Ducourthial. *New Operators for Computing with Associative Nets.* Proceedings of the 5th International Colloquium on Structural Information and Communication Complexity (SIROCCO'98), Amalfi, Italia, 1998.

[18] B. Ducourthial and Sébastien Tixeuil. *Self-stabilizing Global Computations with r-operators.* Proc. of the 2nd International Conference On Principles Of DIStributed Computing (OPODIS'98), pp. 99–113, Hermes, 1998.

[19] D. El Baz, P. Spiteri, J.-C. Miellou and D. Gazen. *Asynchronous Iterative Algorithms with Flexible Communications for Non-Linear Network Flow Problems.* Journal of Parallel and Distributed Computing (JPDC), Vol. 38, pp. 1–15, 1996.

[20] M. Gondran. *Algèbre linéaire et cheminement dans un graphe.* Revue Française d'Automatique, d'Informatique et Recherche Opérationnelle (RAIRO), Informatique Théorique, Vol. 9, No. 1, pp. 77–99, Janurary 1975.

[21] M. Gondran and M. Minoux. *Graphes et Algorithmes.* No. 37 in Collection de la DER de EDF, Eyrolles, 1979. Also published as *Graphs and Algorithms.* John Wiley and Sons, NewYork, 1986.

[22] A. Israeli and M. Jalfon. *Uniform Self-stabilizing Ring Orientation.* Information and Computation, Vol. 104, pp. 175–196, Academic Press, 1993.

[23] J. Miellou. *Algorithmes de relaxation chaotique à retards.* Revue Française d'Automatique, d'Informatique et Recherche Opérationnelle (RAIRO), pp. 55–82, April 1975.

[24] M. Minoux. *Structure algébrique généralisée des problèmes de cheminement dans les graphes : théorèmes, algorithmes et applications.* Revue Française d'Automatique, d'Informatique et Recherche Opérationnelle (RAIRO), Recherche Opérationnelle, Vol. 10, No. 2, 1976.

[25] M. Schneider. *Self-stabilization.* ACM Computing Surveys, vol. 25, pp. 45-67, 1993.

[26] M. Schneider. *Flow Routing in Computer networks.* Ph.D. Dissertation, Department of Computer Sciences, The University of Texas at Austin, December 1997.

[27] G. Tel. *Topics in Distributed Algorithms*, volume 1 of *Cambridge International Series on Parallel Computation.* Cambridge University Press, Cambridge, UK, 1991.

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

[29] M.-S. Tsai and S.-T. Huang. *A Self-stabilizing Algorithm for the Shortest Paths Problem with a Fully Distributed Demon.* Parallel Processing Letters, Vol. 4, No. 1&2, pp. 65–72, 1994.

[30] A. Üresin and M. Dubois. *Sufficient conditions for the convergence of asynchronous iterations.* Parallel Computing, Vol. 10, pp. 83–92, 1989.

[31] A. Üresin and M. Dubois. *Parallel Asynchronous Algorithms for Discrete Data.* Journal of the Association for Computing Machinery, Vol. 37, No. 3, pp. 588–606, july 1990.

[32] A. Üresin and M. Dubois. *Effects of asynchronism on the convergence rate of iterative algorithms.* Journal of Parallel and Distributed Computing (JPDC), Vol. 34, pp. 66–81, 1996.