Analyze of Probabilistic Algorithms under Indeterministic Scheduler

Joffroy Beauquier
Univ. Paris-Sud, LRI, CNRS
91405 Orsay Cedex, France
jb@lri.fr

Colette Johnen
Univ. Paris-Sud, LRI, CNRS
91405 Orsay Cedex, France
colette@lri.fr

Abstract

In a distributed system, the environment is described by the scheduler (also called adversary or demon). Through an example related to stabilization, we show that a formal proof that does not use a formal definition of a scheduler is pointless. As a matter of fact, we show that the same algorithm, according to the scheduler, can be either correct or incorrect and in the cases where it is correct, can have different complexities. The paper is an attempt to better understand the meaning of proving a probabilistic algorithm in a indeterministic environment.

1. Introduction

Perhaps the main reason for the difficulty of understanding the behavior of distributed programs is their inherent indeterminism. Distributed systems are loosely coupled in the sense that the relative speeds of their local activity is usually not known in advance and in most cases totally unpredictable. Execution times and message delays may vary substantially for several repetitions of the same algorithm. But a distributed program has to be correct despite this uncertainty. In other words, the proof of a distributed program must take into account the indeterminism created by the environment. That is the reason why the notion of adversary, also called demon or scheduler, has been introduced. The words adversary or demon refer to a hostile environment, that would try to disturb program executions, the idea being that to prove the correction of a program in the "worst" environmental situation, guarantees its validity in a less severe surrounding. Scheduler refers to the fact that independent events must appear in some given order In this paper, we will use equivalently demon or scheduler. Among the "tasks" devoted to the scheduler appear the choice of the speeds of processes, the delays for message transmission and the scheduling of tasks. For instance, if several processes are able to take a step, the scheduler is the entity that chooses which ones will be activated. In the context of fault tolerance, the scheduler chooses which process will fail and at what time, or which messages will be lost or delayed. Several impossibility results like in [8] or [1] use in their proofs such schedulers. As we will show it in this paper, the correction of a distributed program is meaningless if the external indeterminism is not considered. A consequence is that if the program is proved to be correct in an abstract model, the scheduler has to be part of this model. A way to deal with an unpredictable environment is to make it more predictable. That is the case when it is assumed that the scheduler acts probabilistically [7]. For instance when several processes can take a step simultaneously, there is some fixed probability for each one to take effectively the step. Or each time a message is sent, there is a small probability that it becomes lost. Some could argue that such a scheduler model is sufficient for most practical situations and that in real life environment follows statistical laws, but we think that, for several reasons, such a point of view is unsatisfactory.

The first reason is that a probabilistic scheduler cannot be invoked for proving without changing the specification of the problem. Take the consensus problem and one of its specifications given in [10]. Suppose that each message can be lost with some probability. Bracha and Toueg developed a solution under this assumption but what they solved in [7] is a probabilistic specification of the asynchronous consensus, not the original specification (which is impossible to solve). This remark is quite general and involves that all that can be proved with a probabilistic scheduler are programs solving probabilistic specifications.

The second reason is purely theoretical. A probabilistic scheduler is a very poor adversary, that can be deceived so easily. Some program are able to resist to very powerful adversaries and it is a challenge to always get a result as strong as possible.

The third reason is that we think that, in real life, some schedulers are not probabilistic at all, in the sense that the coming out of events does not obey simple independent probabilistic laws. The point is obvious if malicious adversaries are considered. A software error will always cause

the same failure each time the bad portion of code is executed and, if at some point of an execution, some messages become very slow, the chance that following messages be also delayed will increase. In fact that some bad events are correlated, rather than follow independent probability laws as expressed in [10] : "However we should keep in mind that this assumption (the bounded number of failures) is somewhat problematic : in most practical situations, if the number of failures is already large, then it is likely that more failures will occur.". For all these reasons, a model including distributed (possibly probabilistic) programs and indeterministic schedulers is absolutely essential. Such a model has been presented

In this paper we address the following problem : given a distributed program, how to define the class of schedulers for which the program is correct, i.e. satisfies its specification. This problem is ticklish, because if some programs are either correct or incorrect for any scheduler, some others are not. We adopt here a slightly different presentation, in order to be able to put precisely a borderline between "good" and "bad" schedulers for a given distributed program.

Our second goal is to demonstrate how the presented framework allows a precise computation of complexity measures in term of expectation. For the sake of clarity, we chose a very simple example, with a probabilistic specification, to make clear that a powerful indeterministic scheduler can indefinitely postpone the realization of the specification, while a feeble one cannot. This example is closely related to a self-stabilization problem, namely the token circulation, and can be considered as a typical example of how to prove and compute the complexity of self-stabilizing distributed algorithms. The example is a randomly delayed circulation of two tokens on an unidirectional ring. In the initial configuration, there are two tokens on the ring, held by two different processes. There is no assumption on the processes that initially hold the tokens. Only a process holding a token can take a step. A step consists in tossing a coin (probability 1/2 for head and tail) and if head to transmit the token. For taking a step, a process must be chosen by the scheduler. The scheduler must choose processes among those holding a token. Finally, the specification is that one of the initial tokens catches up with the other with probability one. To the question whether the above system meets or not the specification, most people would answer positively. The idea is that if one of the token does not move because it always "draws" tail and if the other always (or at least a number of time at most the size of the ring) "draws" head, the second token will catch up with the first. Because this scenario can appear at any time with a fixed probability, we are done. As you have noticed, no scheduler appears in this reasoning, and it is why it is incorrect. The paper (we hope) will make this point clearer.

2. Model

In a distributed system, the topology of the network of processes is usually given under the form of a communication graph $G = (V, E)$, where the set $V = \{1, ..., N\}$ corresponds to the processes set. There is an edge between two processes p and q when the both processes can communicate directly; they are say neighbors. Communication among neighbors processes is carried out by shared variables. All neighbors of p can read p's shared variables.
The state of a process is the collection of values of the process's variables (internal or shared). A configuration c of a distributed system is the $|V|$-tuple of all process states. The *local configuration* of p is the part of a configuration that is "seen" by the process p (i.e. p's state and the value of the shared variables of op's neighbors).
Each process executes its code. A process code is a finite set of guarded rules: (i.e. label:: guard → action). The guard of a rule on p is a boolean expression involving p local configuration. The action of a p rule updates the p state. In case of randomize action, the obtained state is computed according the value of its random variable; thus several states may be obtained (they have distinct probabilities).
A process p is *enabled* at a configuration c, if a rule guard of p is satisfied in c. We assume that no process of a distributed system satisfies the guards of two actions in the same configuration. Thus, at most one guard is enabled per process in any configuration. Let c be configuration, $Enabled(c)$ denoted the set of enabled processes at c.
Let c be a configuration, and CH be a subset of $Enabled(c)$. We denote by $< c : CH >$ the set of configurations that are reachable after that all processes of CH perform simultaneously an action. A computation steps is defined by three elements : (1) a configuration, and (2) CH a set of enabled process at c, and (3) a configuration of $< c : CH >$. In the case of a deterministic protocol $< c : CH >$ contains a single configuration.
In the case of a randomized protocol, the computation step output depends on the output of process action. Thus $< c : CH >$ contains several configurations, at each of them it is associated a probability. Therefore, a computation step has another characteristic element : the probabilistic value associated to the computation step. This value depends on the probabilistic laws of the executed rules. Each process has a random variable. The output of an action of a process p depends on the value of p random variable. The process random variables are independent, thus the output of a p action is independent of the output of an action of another process. The probability of a computation step is the product of probability of every output of actions that have been performed during the computation step.

Definition 1 *The probabilistic value associated to a computation step, $pr(c, CH, c')$ is defined by:*

$pr(c, CH, c') = \prod_{p \in CH} pr(X_p = val_p)$, *where* X_p *is the random variable of the process* p, val_p *is a value of* X_p, *and* c' *is the obtained configuration after that all processes of* CH *have set their* X_p *variable to* val_p *and have performed their enabled rule.*

It is easy to prove that $\sum_{c' \in <c:CH>} pr(c, CH, c') = 1$.
A *computation* is a sequence of consecutive computation steps. A computation is *maximal*, if the computation is either infinite, or finite and the final configuration is a deadlock. Let h be a finite computation, *Enabled(h)* denoted the set of enabled processes at the last configuration of h.

Example 1 *Let us study the distributed system* $DS4_{CTC}$, *defined as a ring of* 4 *processes named* $(p1, p2, p3, p4)$ *executing the protocol CTC (its code is given in protocol 1). Let us denote* $[p2p4]$ *a configuration where only the processes* $p2$ *and* $p4$ *are enabled.* $< [p3p4] : \{p3, p4\} >$ *contains four configurations respectively denoted* $[p3p4]$ *(no process has changed its state),* $[p4]$ *(only* $p3$ *has changes its state, now only* $p4$ *is enabled),* $[p1p3]$ *(only* $p4$ *has changed its state, now* $p1$ *and* $p3$ *are enabled), and* $[p1p4]$ *(both processes change their state, now* $p1$ *and* $p4$ *are enabled). The probability associated with each of this computation step is the same :* $1/4$.

Protocol 1 token circulation on anonymous and unidirectional rings: CTC

Shared variable on p:
  v_p is a variable taking value in [0, m_N -1].

Random Variable on p:
  $rand_bool_p$ taking value in $\{1, 0\}$.
  Each value has a probability 1/2.

Predicate on p:
  $Token_p$:: $v_p - v_{lp} \neq 1 \bmod m_N$
macro on p:
  $Pass_Token_p$:: $v_p := (v_{lp} + 1) \bmod m_N$
Action on p:
  $\mathcal{A}$:: $Token_p \longrightarrow$
    if ($rand_bool_p$ = 0) then $Pass_Token_p$;

A distributed system in the case of probabilistic algorithm can be seen as a play between two players. One of the players is the algorithm (the probabilistic player) and the other one is the scheduler (the indeterministic player). The scheduler choices the enabled processes that will do an action; then, the algorithm computes the random outcome. A Markovien Decision Process (MDP). is the mathematical framework for modeling this kind of plays.

Policy To be able to talk about the probability of computation, we have to associate with each computation a probability measure. The probability of a computation depends on how the enabled processes have been indeterministically chosen along the computation. To represent these indeterministic choices, we use the concept of deterministic policy. Informally, a deterministic policy defines the acting processes knowing the history of the distributed system (the computation steps previously performed).

Definition 2 (Deterministic Policy) *Let* DS *be a distributed system. A deterministic policy* η *of* DS *is a set of conditional probabilities* $Q_\eta(CH|h)$, *such that for all finite computations* h, *we have*
$Q_\eta(CH|h) \in \{0, 1\} \; \forall \, CH \subseteq Enabled(h)$ *and*
$\sum_{CH \subseteq Enabled(h)} Q_\eta(CH|h) = 1$ *if* $Enabled(h) \neq \emptyset$.

Remark 1 *Let* η *be a deterministic policy of* DS. *Let* h *be finite computation of* DS. *It exists at most a single subset of* $Enabled(h)$, CH, *such that* $Q_\eta(CH|h) = 1$.

Definition 3 (Weight of finite computations under η**)** *Let* DS *be a distributed system. Let* η *be a deterministic policy of* DS. *Let* $h = st_1 st_2 ... st_n$ *be the sequence of* n *consecutive computation steps where* $st_k = (c_k, CH_k, c_{k+1})$. *The weight associated at* h *by* η *is*

$$Weight_\eta(h) = \prod_{k=0}^{n-1} pr(st_k) Q_\eta(CH_k | st_0 ... st_{k-1})$$

Example 2 *We define the policy* η_c *of* $DS4_{CTC}$ *as follow: if* $Enabled(h)$ *contains* pi *but not* pj *where* $j < i$ *then* $Q_{\eta_c}(CH|h) = pi$. *The weight of any computations of length* n *under* η_c *is* 2^{-n} *or* 0. *We define the policy* η_d *of* $DS4_{CTC}$ *as follow: if (1)* $Enabled(h)$ *contains* p *and it the right neighbor then* $Q_{\eta_d}(CH|h) = p$, *otherwise if (2)* $Enabled(h)$ *contains* pi *but not* pj *where* $j < i$ *then* $Q_{\eta_d}(CH|h) = pi$. *The weight of any computations of length* n *under* η_d *is* 2^{-n} *or* 0.

The number of deterministic policies is unbounded but also unenumerable.

Probabilistic Spaces A strategy can be seen as the set of games (computations) between the protocol and an oracle. All games (computations) have the same start (i.e. the same initial configuration). Along a game, the oracle chooses the next transition that the protocol will perform, the output configuration of this transition is randomly computed by the protocol. In all these games, the oracle policy is the same: the oracle choice is only function of the history of the current game (i.e. the series of computation steps that has been performed).

Definition 4 (Strategy)
Let DS be a distributed system and η be a deterministic policy of DS.
Let e be a computation of DS. If for every prefix of e, h, we have $Weight_\eta(h) > 0$ then we said that e adheres to the policy η.
The strategy $st = (c, \eta)$ *of DS is the set of maximal computations of DS adhering to η and whose the initial configuration is c.*

Definition 5 (cylinder sets) *Let DS be a distributed system and $st = (c, \eta)$ be a strategy. Let h be a finite computation. The basic cylinder C_h of st contains all computations of st having the prefix h: $C_h = \{hw \in st\}$*
The probability of the basic cylinder C_h of $st = (c, \eta)$, is defined as $P_{st}(C_h) = Weight_\eta(h)$ if the initial configuration of h is c or 0 otherwise.

Example 3 *In the strategy $([p2p4], \eta_d)$ of $DS4_{CTC}$ presented in the Figure 1, the ring always keeps two enabled processes (i.e. the ring has two tokens). In the strategy $([p2p4], \eta_c)$ of $DS4_{CTC}$ presented in the Figure 2, the probability to reach a configuration with a single token is 1, because the probability to reach such a configuration in less than $2k$ computation steps in the strategy ([p2p4], η_c) is greater than $1 - (3/4)^k$.*

In a strategy st, the cylinder sets are measurable sets. Let B_{st} be the smallest algebra of subsets of st, that contains all the basic cylinder sets of st and that is closed under complement and countable unions and intersections. This algebra is called the Borel σ-$algebra$ of the st's basic cylinder sets. All elements of B_{st} are measurable sets of computations. It is well-known that the triple (st, B_{st}, P_{st}) defines a probabilistic space on st.

Scheduler Basically, a scheduler is intended to be an abstraction of the external indeterminism. Because the effect of the environment is unknown in advance, the scheduler notion must be able to formalize any external behavior. Defining a scheduler in some operational way - at that point of the computation the scheduler has such or such choice - raises the problem to define exactly in function of what the choice is made. If the choice depends of the actual configuration and of the history, the generality of the scheduler is restricted. How, for instance, express that the scheduler must be fair. That is the reason why in a deterministic system, we define a scheduler as a predicate (subset) on infinite computations. In our framework, the key notion is the policy. Formally, a *scheduler* is completely defined by the set of policy which it may 'execute'.

Definition 6 *Let DS be a distributed system.*
A scheduler D is a set of policies,

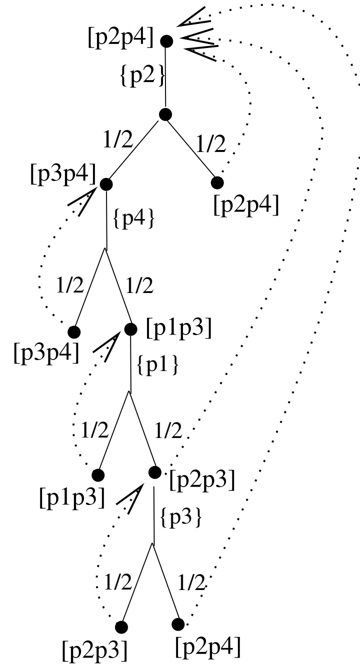

Figure 1. A divergent strategy of $DS4_{CTC}$

The strategies of DS under D is the set of all strategies based on a policy of D.
A schedule is fair if any strategy based on a policy of D contains only fair computations.

Attractor In this section, we define the notion of attractor for probabilistic protocols with respect to the probabilistic model defined in the previous section. The main idea behind these definitions is simple : to analyze a probabilistic distributed algorithm under a scheduler, one has to analyze every policy of the scheduler starting in the initial configuration. Similarly, to analyze a probabilistic self-stabilization algorithm under a scheduler, one has to analyze every policy of the scheduler starting from every configuration.

Notation 1 *Let L be a predicate over configurations. We denote by $\mathcal{EL}$ the set of st computations reaching a configuration that satisfies the predicate L.*

Definition 7 (Convergence) *Let L be a predicate defined on configurations. A distributed system DS under a scheduler D probabilistically converges to L iff : In any strategy st of DS based on a policy of D the probability of the set of computations reaching L is equal to 1. Formally, $\forall st, P_{st}(\mathcal{EL}) = 1$.*

Definition 8 (attractor) *Let L be a predicate defined on configurations. L is an attractor of the distributed system*

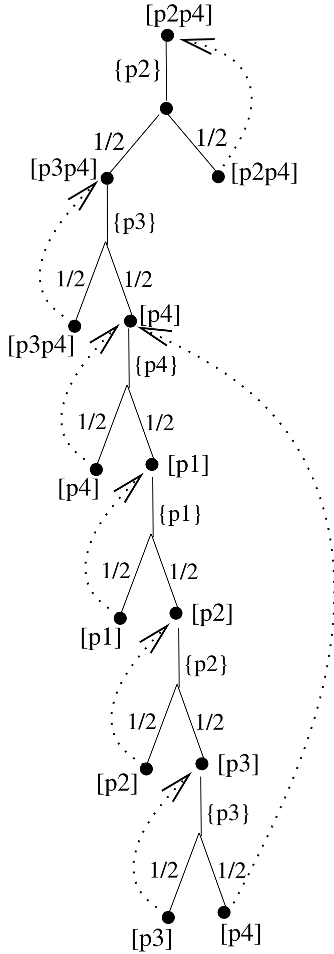

Figure 2. A convergent strategy of $DS4_{CTC}$

DS under the scheduler D iff (1) DS under D probabilistically converges to L and (2) L is a closed predicate.

3 CTC Protocol

We study the randomized token circulation protocol (see protocol 1) on unidirectional anonymous rings designed by Beauquier, Cordier and Delaët in [2]. The space complexity of the Protocol 1 is $O(\lg m_N)$ bits per process where m_N is the smallest integer not dividing N (N being the ring size). Notice that the value of m_N is constant on average. (private communication of Jean Paul Allouche). For example, on odd size rings, 4 (2) bits per process are necessary and sufficient for the token circulation.

A process holds a token iff it is enabled (i.e. $v_p - v_{lp} \neq 1 \bmod m_N$) There is always a token in the ring. Let L be the following predicate over configurations: "there is only one token in the system". The number of tokens cannot increase whatever may happen; thus L is a closed predicate. This protocol is important because it is very simple; and nevertheless, it is a needed layer of more powerful protocol. The protocol 1 was used to design a self-stabilizing leader election on unidirectional anonymous rings that stabilizes under any scheduler [4, 3]. This leader election protocol is optimal in space complexity. The protocol 1 was used to design a fast self-stabilizing token circulation algorithm on unidirectional anonymous rings that stabilizes under any scheduler [9]. This fast token circulation protocol is optimal in space complexity and once the system is stabilized, the fast token reaches every process in N computation steps - N being the ring size - (i.e. the service time is optimal). This fast token circulation is also optimal in space complexity.

In [2], it was proven that the protocol 1 converges under the fair memory k-bounded policies from every initial configuration. In [3], it was proven that the protocol 1 converge under k-bounded policies from every initial configuration. A computation is *k-bounded* only if a process (other than p) performs at most k actions till p stay enabled. A policy η is *k-bounded* if any strategy based on it contains only k-bounded computations. In [5], it has been proven that any fair memory k-bounded strategy is a k-bounded strategy the converse is not true.

Hence, we have illustrated in Figure 1, the protocol 1 does not converge under some policies. These policies contain some unfair computations. Among the "bad" policies there is the following unfair policy, that can be informally described in the following way (but precisely defined using [6]). The oracle chooses the process having a token until the move of this token; then it chooses the process having the other token until the moving of this second token; and so one. If initially the tokens are at distance 2 or more, no one will ever catch up with the other.

Our aim is to determine exactly where is the borderline between the class of fair policies that prevent the convergence and the class of those that do not. In the case of convergence, we want also have some information about the expectation time (in fact number of steps) before convergence.

In the following section, we will analyze this protocol under a large class of fair policies to determine when this protocol converges and when it does not, when the expectation time is bounded and when it is not.

For the sake of clarity we will not start the protocol from any configuration, but only from the configurations in which there are exactly two tokens in the ring, the distances between them being strictly greater than 1.

4 Rotating Policies

In an unidirectional ring, The *distance* from process p to process q is measured starting from p and moving to the right until q is reached. The distance from p to q is not the

distance from q to p.
Because there are only two tokens (but a more general treatment would be feasible) a class of fair policies can be described as what we call rotating policies. Roughly speaking, in a rotating policy the oracle chooses to try to move one of the token tr_1 times, then the other tr_2 times, then the first token tr_3 times and so one. Because the oracle is an adversary and tries to avoid the token merging, as soon as one of the token moves it deals with the other. Then, a rotating policy is completely defined via an infinite sequence of positive integers $tr_1, tr_2, tr_3...$ where tr_i is the maximal duration (i.e. the maximal number of computation steps) of the ith turn. At each computation step of the 2i+1th (resp. 2i+2th) turn, only the process having the token $T1$ (resp. $T2$) performs its action; thus a 2i+1th (resp. 2i+2th) turn is said to be a $T1$ (resp. $T2$) turn. The ith turn (a Tx turn) stops and the i+1th turn begins (i) after that the Tx token has moved or (ii) after tr_i consecutive actions of the process having the Tx token where the token does not move. In a k-bounded strategy [3], the used policy is a rotating one in which the duration of each turn is inferior to $k + 1$.

Definition 9 (rotating strategy) *Let $st = (c_0, \eta)$ be a strategy. st is a rotating strategy if, in c_0 there are two tokens at distance greater than* 1*, and (2) η is a rotating policy.*

We denote by ϵ_i the probability in st that a token catches up with the other token during the ith turn. Once, the two tokens have merged, a legitimate configuration is reached.
The algorithm may perform as many trials he wants, thus $P_{st}(\mathcal{EL}_{st}) = \sum_{i=1}^{\infty}(\epsilon_i \times \prod_{j=1}^{i-1}(1 - \epsilon_j))$.

Summary of results In the initial configuration $c0$, there are two tokens in the ring. The token distances are strictly greater than 1.
Let $k1$, $k2$, $k3$, $k4$ and M be positive integers. Let α be a positive real.
Let $st = (c_0, \eta)$ be the rotating strategy where $\eta = tr_1, tr_2, tr_3....$
The first table presents our results about the convergence in st according to tr_i values. We have defined the borderline between the rotating strategies that do converge and the other ones (i.e $tr_i = log_2(k1(i + k2))$).

Definition of rotating strategy	Convergence
$\forall i \geq 1, tr_i > \log_2(C_{1+\alpha}(i+1)^{1+\alpha})$	no
$\forall i \geq M, tr_i \leq k1$	yes
$\forall i \geq M, tr_i \leq \log_2(k1(i+k2))$ $tr_i - tr_{i+1} \geq -k3$	yes

The second table presents our results about the expectation time in st according to tr_i values (when the algorithm converges). We have not found the exact borderline between an unbounded and a bounded expectation time. The main difficulty is to compute precisely the value ϵ_i (i.e. the probability of convergence during the ith turn). Nevertheless we know that the borderline is located between $log_2(k1(i + k2))$ and $log_2((i + 1))^{1/2}$.

Definition of rotating strategy	Expectation time
$\forall i, tr_i \leq k1$	$\leq C + 2^{(k+1)\times(n-1)}$ $C = 2n - 4$
$\forall i \geq M,$ $tr_i \leq \log_2(k1(i+k2))^{1/2};$ $tr_i - tr_{i+1} \geq -k3$	bounded
$\forall i \geq M,$ $\log_2(k1(i+k2))^{1/2} < tr_i$ $tr_i < \log_2(i+1)$ $tr_i - tr_{i+1} \geq -k3$	???
$\forall i \geq M,$ $\log_2(i+1) \leq tr_i$ $tr_i \leq \log_2(k1(i+k2))$ $tr_i - tr_{i+1} \geq -k3$	unbounded

References

[1] E. Anagnostou and V. Hadzilacos. Tolerating transient and permanent failures. In *WDAG93 LNCS:725*, pages 174–188, 1993.

[2] J. Beauquier, S. Cordier, and S. Delaët. Optimum probabilistic self-stabilization on uniform rings. In *WSS95*, pages 15.1–15.15, 1995.

[3] J. Beauquier, M. Gardinariu, and C. Johnen. Randomized self-stabilizing and space optimal leader election under arbitrary scheduler on rings. *Distributed Computing*, 20(1):75–93, 2007.

[4] J. Beauquier, M. Gradinariu, and C. Johnen. Memory space requirements for self-stabilizing leader election protocols. In *PODC99*, pages 199–208, 1999.

[5] J. Beauquier, C. Johnen, and S. Messika. All k-bounded policies are equivalent for self-stabilization. In *SSS06, LNCS: 4280*, pages 82–94, 2006.

[6] J. Beauquier, C. Johnen, and S. Messika. Brief announcement: Computing automatically the stabilization time against the worst and the best schedulers. In *DISC06, LNCS:4167*, pages 543–447, 2006.

[7] G. Bracha and S. Toueg. Asynchronous concensus and broadcast protocols. *Journal of ACM*, 32(5):324–340, 1985.

[8] M. H. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[9] C. Johnen. Service time optimal self-stabilizing token circulation protocol on anonymous unidrectional. In *SRDS 2002*, pages 80–89, 2002.

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