# Analyzing Expected Time by Scheduler-Luck Games*


Shlomi Dolev[†] Amos Israeli[‡] Shlomo Moran[§]


November 27, 1994


**Abstract**

In this paper we introduce a novel technique, the *scheduler luck game* (in short *sl-game*) for analyzing the performance of randomized distributed protocols. We apply it in studying uniform self-stabilizing protocols for leader election under read/write atomicity. We present two protocols for the case where each processor in the system can communicate with all other processors and analyze their performance using the *sl-game* technique.



1991 Mathematics Subject Classification: 68M10,68M15,68Q22
Keywords & Phrases: Self Stabilizing Systems, Leader Election, Randomized Distributed Algorithms.


---

*The results presented in this work were first reported, among other results, in the Proceedings of the 5-th International Workshop on Distributed Algorithms (1991). An extended version of this paper appear in Technical Report 807, Computer Science Dept., Technion (1994).


[†]Department of Computer Science, Texas A&M University, College Station, TX 77843. Supported in part by TAMU Engineering Excellence funds and NSF Presidential Young Investigator Award CCR-9158478. Part of this work was done while this author was at the Technion — Israel.

[‡]Department of Electrical Engineering, Technion — Israel.

[§]CWI, P.O. Box 4079, 1009 AB Amsterdam, the Netherlands, and Department of Computer Science Technion — Israel.

# 1 Introduction

The analysis of the time complexity (measured in round as defined in the sequel) of distributed randomized protocols is often a very complicated task. This task is specifically hard in self stabilizing protocols, in which no assumption is made on the initial state of the processes. In this paper we propose a useful tool, called the *sl-game method*, for proving upper bounds on the time complexity of randomized distributed protocols, and demonstrate it on self stabilizing protocols.

The players of the game are called *scheduler* (or adversary) and *luck*, their opposing goals are to prevent the protocol from fulfilling its task and to help it fulfilling its task, respectively. In each turn of the game the scheduler chooses the next processor to be activated which then makes a step. If during this step the activated processor tosses a coin, then *luck* may *intervene* — i.e. *luck* may determine the result of the coin toss. Note that *luck* may choose not to intervene even though the activated processor tosses a coin, in such a case each possible result (*head* or *tail* of the coin toss have equal probability to occur). Both players are assumed to have unlimited computational resources, and their decisions are based on the history of the game so far. In other words: *scheduler* can be looked at as a function from executions to the set of processor numbers while *luck* can be looked at as a function from executions to the set {*head, tail, don't care*}. When the step of the activated processor ends a new turn begins. We prove that when *luck* has a winning strategy for the game in expected number of at most $r$ rounds and with $f$ interventions then the protocol achieves its goal within $r2^f$ expected number of rounds.

A restricted version of this game appears in [Ab-88] for a different model and without any analysis. In the game of [Ab-88] *luck* intervene whenever a coin is tossed. In contrast, in our *sl-game luck* may intervene only when the result is "critical". Thus, our game is a generalization of the game presented in [Ab-88] that achieves tighter bounds on the time complexity of randomized protocols. Note that a proof of correctness and performance of a protocol by competing with an adversary appears in the literature e.g. [CC-85]. However, here we introduce an opposing player, namely *luck*, that (roughly speaking) helps the protocol in critical stages. Then, based on the strategy of *luck* for helping the protocol, we analyze the expected time for the algorithm to achieve its task.

We apply the *sl-game* to analyze the time complexity of uniform self-stabilizing protocols for leader election. Leader-election is a fundamental task in distributed computing. Roughly speaking, a protocol that solves this task requires that when its execution terminates, a single processor is designated as a *leader* and every processor knows whether it is a leader or not. By definition, whenever a leader-election protocol terminates successfully, the system is in a non-symmetric global state (henceforth we use the term *configuration* for a system global state). Any leader-election protocol that has a symmetric initial state requires some means of symmetry breaking. In *id based* systems each processor has a unique identifier called the processor's *id*, hence the system has no symmetric configuration. In *uniform*[1] leader-election protocols, all processors are identical, the initial state is symmetric and symmetry is broken by randomization.

A distributed system is *self-stabilizing* if it can be started in any *possible* configuration. Once started the system reaches a *legitimate* configuration, by itself without any kind of outside intervention. The self-stabilization property makes the system tolerant to faults in which processors crash and then recover spontaneously in an arbitrary state. When the intermediate period between one recovery and the next crash is long enough, the system stabilizes.

A protocol is called *semi-uniform* if it has two kinds of processors: A unique predetermined processor of one type and all other processors are of the other type. The unique processor serves as a leader and prevents the existence of symmetric configurations. Most of the self-stabilizing protocols presented so far are semi-uniform. If one could run a semi-uniform protocol on a uniform system, the repertoire of uniform self-stabilizing protocols would be considerably enlarged. Let $\mathcal{PR}$ be an arbitrary semi-uniform, self-stabilizing protocol. To run $\mathcal{PR}$ on a uniform system we employ a uniform, self-stabilizing, leader-election protocol and compose it with $\mathcal{PR}$, using *fair protocol composition*— a technique presented in [DIM-93]. The resulting protocol is a uniform protocol whose behavior is identical to the behavior of $\mathcal{PR}$. Thus, a uniform self-stabilizing leader election protocol enables any semi-uniform, self-stabilizing protocol to be converted to a uniform version of the same protocol.

The *stabilization time* of a self-stabilizing protocol is the maximal time that the system takes to reach a legitimate configuration where the maximum is taken over all possible executions. We consider stabilization time a very important complexity measure and carefully analyze our protocols' stabilization time, using the *sl-game* technique mentioned above.

We present two uniform, self-stabilizing, leader-election protocols for complete-graph systems. The first protocol is a simple minimum space protocol in which each processor uses *one* shared bit to elect a leader; the subtlety of self-stabilizing systems is demonstrated by showing a somewhat surprisingly exponential lower bound on the time complexity of this protocol. The correctness of the minimum space protocol assumes *coarse atomicity* (coarse and fine atomicity are defined in the next section). The second protocol uses space linear in $n$, the number of processors in the system; its time complexity is shown to be, by using the *sl-game* method, $O(n \log n)$. This protocol is correct under *fine* atomicity.

The rest of this paper is organized as follows: In Section 2, we present the formal model and requirements for uniform, self-stabilizing protocols. In Section 3 we present

[1] Uniform systems are also referred to as *anonymous*.

and analyze the *sl-game* method. Section 4 presents two self-stabilizing leader election protocols, and use the *sl-game* technique to analyze them.

## 2 Model and Requirements

A *uniform distributed system* consists of $n$ processors denoted by $P_1, P_2, \cdots, P_n$. Processors are *anonymous*, they do not have identities. The subscript $1, 2, \cdots, n$ are used for ease of notation only. Each processor communicates with all other processors using a single writer, multi reader register which is serializable with respect to read and write actions. For the sake of clarity, we assume that every processor knows the exact contents of the register that it is writing to[2].

For ease of presentation, we regard each processor as a *CPU* whose program is composed of *atomic steps*. An atomic step of a processor consists of an internal computation followed by a terminating action. Under *fine* atomicity, the terminating actions are **read**, **write** and **coin toss**. Under *coarse* atomicity, a coin-toss is considered an internal action and does not terminate the atomic action containing it. We assume that the state of a processor fully describes its internal state and the value written in its register. Denote the set of states of $P_i$ by $S_i$. A *configuration*, $c \in (S_1 \times S_2 \times \cdots S_n)$, of the system is a vector of states of all processors.

Processor activity is managed by a scheduler. In any given configuration, the scheduler activates a single processor which executes a single atomic step. To ensure correctness of the protocols, we regard the scheduler as an adversary. The scheduler is assumed to have unlimited resources, and it chooses the next activated processor *on line*, using the full information on the execution so far. An *execution* of the system is a finite or an infinite sequence of configurations $E = (c_1, c_2, \cdots)$ such that for $i = 1, 2, \cdots$, $c_{i+1}$ is reached from $c_i$ by a single atomic step of some processor. A *fair execution* is an infinite execution in which every processor executes atomic steps infinitely often. A scheduler $S$ is *fair* if for any configuration $c$, with probability 1, an execution starting from $c$ in which processors are activated by $S$ is fair.

In a distributed asynchronous system, each processor may operate at any non constant rate and different processors might be slow in different parts of the execution. On the other hand, completion of a task may require that each processor executes some number of steps. This is especially true in self-stabilizing systems: Consider for instance a self-stabilizing mutual exclusion protocol in which two processors are "stuck" inside the critical section. Regardless of the number of steps taken by all other processors, the system will not stabilize until at least one of the stuck processors gets out of the critical

[2]One may assume that every processor refreshes the contents of its register periodically.

section. Thus, the notion of *fairness* is very important in self-stabilizing systems[3] The following definition of round complexity attempts to give a complexity measure in which the unfair behavior of the adversary is neutralized, by capturing the rate of action of the slowest processor in any segment of the execution. Given an execution $E$, we define the *first round* of $E$ to be the minimal prefix of $E$, $E'$, containing atomic steps of every processor in the system. Let $E''$ be the suffix of $E$ for which $E = E' \circ E''$. The second round of $E$ is the first round of $E''$, and so on. For any given execution, $E$, the *round complexity* (which is sometimes called the execution time) of $E$ is the number of rounds in $E$. Under this definition, the time to complete a single round is unbounded and depends on the fairness of the adversary. Any self-stabilizing application that uses our protocol as a subroutine would probably also require fair behavior to stabilize and its complexity will be proportional to the stabilization complexity of our protocol.

We proceed by defining the self-stabilization requirements for randomized distributed systems. A behavior of a system is specified by a set of executions. Define a *task* $LE$ to be a set of executions which are called *legitimate executions*. A configuration $c$ is *safe* with respect to a task $LE$ and a protocol $\mathcal{PR}$ if *any* fair execution of $\mathcal{PR}$ starting from $c$ belongs to $LE$. Finally, a protocol $\mathcal{PR}$ is *randomized self-stabilizing* for a task $LE$, if starting with any system configuration and considering any fair scheduler, the protocol reaches a safe configuration within an expected number of rounds which is bounded by some constant $C$. (The constant $C$ may depend on $n$, the number of processors in the system).

# 3 Scheduler-Luck Games

In this section we introduce a new method to analyze randomized distributed protocols, by using a full information two player game, called *sl-game*. An *sl-game* is a triplet $\mathcal{G} = (\mathcal{PR}, \mathcal{I}, \mathcal{F})$ where $\mathcal{PR}$ is a protocol, $\mathcal{I}$ is a set of initial configurations and $\mathcal{F}$ is a set of final configurations. In the context of self-stabilization $\mathcal{I}$ is the set of all possible configurations and $\mathcal{F}$ is the set $\mathcal{C}$ of configurations which are safe w.r.t. some task $LE$ and the protocol $\mathcal{PR}$, but this is not essential for using the method. The players of $\mathcal{G}$ are called *scheduler* (or adversary) and *luck*, their opposing goals are to prevent $\mathcal{PR}$ from reaching a configuration in $\mathcal{F}$ and to help it to reach such a configuration, respectively.

The states of $\mathcal{G}$ are system configurations of the protocol $\mathcal{PR}$; each turn of $\mathcal{G}$ starts from some configuration $c$, and in each turn the scheduler chooses the next activated processor which then makes an atomic step. If, during this step the activated processor

[3] We should note that for a restricted class of tasks self-stabilization is possible even if some processors permanently halt e.g. [DW-93].

tosses a coin, then *luck* may intervene and determine the result of the coin toss. If *luck* chooses to intervene and determines the result of the coin toss, say to *head*, then we say that *luck fixed* the result of this coin toss to be *head*. Both players are assumed to have unlimited computational resources, and their decisions are based on the history of the game so far. In other words: *scheduler* can be looked at as a function from executions to the set of processor numbers while *luck* can be looked at as a function from executions to the set {*head, tail, don't care*}. When the atomic step is completed, a new system configuration $c'$ is reached from which a new turn begins. Each execution of $\mathcal{G}$ corresponds naturally to an execution of the protocol $\mathcal{PR}$. The execution of $\mathcal{G}$ terminates (if at all) when the system reaches a configuration in $\mathcal{F}$. The scheduler in an *sl-game* is required to be fair, but otherwise it is arbitrary. Our main result in this section is to establish a relationship between winning strategies for *luck* and the expected round complexity of $\mathcal{PR}$. We begin the discussion with some definitions:

**Definition 1:** Let $T$ be a directed tree: $L$ is a length function for $T$ if for every node $u$ in $T$, $L(T,u)$ is a nonnegative integer, and it satisfies the following properties:

1. If $u$ is the father of $v$ in $T$ then $L(T,u) \leq L(T,v)$.
2. If $u$ and $v$ have the same father in $T$ then $L(T,u) = L(T,v)$.
3. If $T'$ is obtained from $T$ by addition of a leaf $u$ then for every node $v \neq u$ $L(T,v) = L(T',v)$.[4]

**Definition 2:** Let $T$ be a binary tree and $L$ a length function. For each node $u$ in $T$, let $p_T(u)$ be the probability to reach $u$ in $T$ in a random walk from the root along directed edges. In other words, $p_T(u) = 2^{-b}$, where $b$ is the number of nodes with two sons on the path from the root to $u$. The *characteristic probability of* $T$, $\overline{P}(T)$, is the sum $\sum p_T(u)$, taken over all the leaves $u$ of $T$. The *characteristic length of* $T$, $\overline{L}(T)$, is the sum $\sum L(T,u) \cdot p_T(u)$, taken over all the leaves $u$ of $T$.

Note that if $T$ is finite then $\overline{P}(T) = 1$, and that if $\overline{P}(T) = 1$ then $\overline{L}(T)$ is the expected length of a leaf in $T$.

**Lemma 1:** Let $T$ and $T'$ be two binary trees and let $L$ be a length function. If $T'$ is derived from $T$ by addition of a new leaf as a son of a non-leaf node of $T$, then $\overline{L}(T) \geq \overline{L}(T')$.

[4] Note that properties 1. and 2. do not necessarily imply 3.: For instance, let $L(T,v)$ be the number of ancestors of $v$ in $T$ which have at least 2 sons; then $L$ satisfies 1. and 2. but not 3.

**Proof:** Let $T$ and $T'$ be as in the statement of the lemma, where $T'$ is obtained from $T$ by adding a new leaf $v$ as a son of a non leaf node $u$ in $T$. Let $U = \{w : w \text{ is a leaf in } T \text{ which is a descendant of } u\}$[5] and let $P = \sum_{w \in U} p_T(w)$.

Observe that every node in $T$ has the same length in $T$ and in $T'$, and every leaf in $T$ which is not in $U$ also has the same probability in $T$ and in $T'$. Also, for every node $w \in U$, $p_{T'}(w) = p_T(w)/2$, and $p_{T'}(v) = P/2$. Thus we have

$$\overline{L}(T') - \overline{L}(T) = L(v) \cdot P/2 - \sum_{w \in U} L(w) \cdot p_T(w)/2.$$

Since every $w \in U$ is a descendant of $u$ and $v$ is a son of $u$, we have that $L(w) \geq L(v)$ for every such $w$. This, and the fact that $\sum_{w \in U} p_T(w) = P$, implies that the right hand side of the above equation is bounded from above by

$$L(v) \cdot P/2 - L(v) \cdot \sum_{w \in U} p_T(w)/2 = 0$$

which implies the lemma. $\square$

**Definition 3:** Let $\mathcal{G} = (\mathcal{PR}, \mathcal{I}, \mathcal{F})$ be an *sl-game*, and let $REACH(\mathcal{I} : \mathcal{F})$ be the set of system configurations which are not in $\mathcal{F}$, and are reachable by executions of $\mathcal{PR}$ that start in a configuration in $\mathcal{I}$ (In particular, $\mathcal{I} \subseteq \mathcal{REACH}(\mathcal{I} : \mathcal{F})$). Given positive finite integers $f$ and $r$, we say that *luck has an $(f, r)$-strategy for $\mathcal{G}$* if for any initial configuration $c_i \in REACH(\mathcal{I} : \mathcal{F})$ and for every scheduler $S$, $\mathcal{G}$ reaches a configuration $c_l \in \mathcal{F}$ in expected number of at most $r$ rounds and with at most $f$ interventions of *luck*.

Throughout the rest of this section, we assume that the game $\mathcal{G} = (\mathcal{PR}, \mathcal{I}, \mathcal{F})$ is fixed and that *luck* has an $(f, r)$-strategy for $\mathcal{G}$.

**Definition 4:** let $E$ be a given execution of $\mathcal{PR}$. The *first block* of $E$, $B_1$, is the prefix of $E$ satisfying one of the following:

1. **good block:** $B_1$ is the minimal prefix of $E$ which is an execution of $\mathcal{G}$ under some scheduler $S$ which ends in a configuration in $\mathcal{F}$ (provided there is such a prefix), or

2. **bad block:** $B_1$ is either (a) an infinite execution of $\mathcal{G}$ under some scheduler $S$ which does not reach a configuration in $\mathcal{F}$, or (b) the minimal prefix of $E$ which is not a prefix of any execution of the *sl-game* $\mathcal{G}$. (In the first case, $B_1 = E$, while in the latter case $B_1$ ends with an atomic operation which contains a coin toss whose outcome is $b \in \{0, 1\}$, where the $(f, r)$-strategy for $\mathcal{G}$ requires *luck* to set the outcome of the coin toss to $\neg b$).

[5]The set $U$ might be infinite.

If $B_1$ is a good block, then when it terminates, the system reaches a configuration in $\mathcal{F}$. Otherwise, $B_1$ is a bad block, and if $B_1$ is finite then when it terminates the system reaches a configuration in $REACH(\mathcal{I} : \mathcal{F})$. If $B_1$ is finite, let $E'$ be the suffix of $E$ defined by $E = B_1 \circ E'$, and let $B_2$ be the first block of $E'$. Again, if $B_2$ is a good block then when it ends the system reaches a configuration in $\mathcal{F}$. Continuing this way, we associate with $E$ a sequence $\mathcal{B} = (B_1, B_2, \cdots)$ of blocks, such that $\mathcal{B}$ is either a (possibly infinite) sequence of bad blocks, or $\mathcal{B}$ consists of $l$ blocks, out of which the first $l-1$ blocks are bad and the $l$th block is good.

Now, let the scheduler $S$ be fixed. For each configuration $c \in \mathcal{I}$, define the *sl-game tree* of $S$ and $luck$ starting from $c$, $GT_c = GT_c(S, luck)$, to be the following directed tree: Each node in $GT_c$ denotes a prefix of an execution of $\mathcal{G}$. The root is the empty execution, and a node $u$ in $GT_c$ has a son $v$ if $v = u \circ (a_i)$, where immediately following the execution defined by $u$ the scheduler $S$ activates a processor which executes the atomic step $a_i$. In case that the processor activated by the scheduler at $u$ tosses a coin and $luck$ does not intervene, $u$ has two sons — one for each possible outcome of the coin toss; otherwise $u$ has one son. If $u$ contains a good block then $\mathcal{G}$ is terminated and $u$ is a leaf. For every node $u$ the probability to reach $u$ in $GT_c$ is denoted by $p_{GT_c}(u)$. This probability is equal to $2^{-b}$, where $b$ is the number of nodes that have two sons in the path from the root to $u$ in $GT_c$.

In a similar fashion, define the *blocks tree* $BT_c = BT_c(S, luck)$ to be a directed tree whose nodes are executions of $\mathcal{PR}$ starting from $c$. $BT_c$ contains all the nodes and links of $GT_c$. In addition, for any node $v$ in $GT_c$ which corresponds to an execution in which $luck$ intervenes, $BT_c$ has one additional son which is a leaf. This additional son represents the execution in which the coin toss result differs from the result fixed by $luck$. As before, we associate with each node $u$ in $BT_c$ a probability $p_{BT_c}(u) = 2^{-b}$ where $b$ is the number of nodes that have two sons in the path from the root to $u$. Note that each leaf in $BT_c$ represents a finite execution of $\mathcal{PR}$ under the scheduler $S$ starting from $c$ and consisting of a single (good or bad) block, and for each such leaf $u$, the probability $p_{BT_c}(u)$ is the probability of the corresponding execution. In order to analyze the performance of our protocols, we say that an execution *$E$ takes $r$ rounds*, if $r$ rounds are initiated in $E$. It is not hard to see that the function, $\overline{L_r}(GT_c, u)$, that counts the number of rounds in (the execution) $u$ is a length function. Note that all internal nodes of $GT_c$ and $BT_c$ belong to $REACH(\mathcal{I} : \mathcal{F})$.

**Lemma 2:** $\overline{L_r}(GT_c)$ is equal to the expected number of rounds in $\mathcal{G}$ that start in configuration $c$.

**Proof:** Recall that there is 1-1 correspondence between the leaves of $GT_c$ and the executions of $\mathcal{G}$ starting at $c$ which are good blocks. Moreover, for each such leaf $u$,

$L_r(T,u)$ is the number of rounds in $u$ and $p_{GT_c}(u)$ is the probability of $u$. First, we show that $\overline{P}(GT_c) = 1$. This follows by the assumption that the expected number of rounds until a configuration in $\mathcal{F}$ is reached in $\mathcal{G}$ is $r$, hence the probability that $\mathcal{G}$ contains infinitely many rounds is zero[6]. Thus, with probability one, a configuration in $\mathcal{F}$ is reached within a finite number of rounds. Therefore, $\overline{P}(GT_c) = 1$. This implies that $\overline{L_r}(GT_c)$ is the expected length of a leaf in $GT_c$. So, $\overline{L_r}(GT_c)$ is also the expected number of rounds in $\mathcal{G}$. In particular, $\overline{L_r}(GT_c) \leq r$. □

We now use the game tree and the block tree of game $\mathcal{G}$ to prove two lemmas which hold for executions of $\mathcal{G}$ scheduled by an arbitrary fair scheduler $S$ starting from an arbitrary initial configuration $c \in \mathcal{I}$.

**Lemma 3:** Let $E$ be an execution of $\mathcal{PR}$. With probability at least $2^{-f}$, the first block of $E$ is good.

**Proof:** Let $U$ be the set of leaves of $BT_c$ which correspond to good blocks. We have to show that $\sum_{u \in U} p_{BT_c}(u) \geq 2^{-f}$. Since $U$ consists of all the leaves in $GT_c$, the proof of Lemma 2 implies that $\sum_{u \in U} p_{GT_c}(u) = \overline{P}(GT_c) = 1$. Let $u$ be an arbitrary node (leaf) in $U$. The path from the root to $u$ in $BT_c$ is identical to the path from the root to $u$ in $GT_c$, and each node $v$ in these paths, which has two sons in $BT_c$ but only one son in $GT_c$, corresponds to an intervention of *luck* in $\mathcal{G}$. Since *luck* has an $(f,r)$-strategy for $\mathcal{G}$ there are at most $f$ such nodes on this path. In other words: if there are $b$ nodes with two sons on the path from the root to $u$ in $GT_c$, then there are at most $b+f$ such nodes on the path from the root to $u$ in $BT_c$. Thus, for each $u \in U$, it holds that $p_{BT_c}(u) \geq 2^{-f} p_{GT_c}(u)$. The lemma follows. □

**Lemma 4:** The expected number of rounds in the first (good or bad) block in an execution starting at any configuration $c$ is at most $r$.

**Proof:** The expected number of rounds in the first block in an execution starting from $c$ is given by $\overline{L_r}(BT_c)$. Thus we have to show that $\overline{L_r}(BT_c) \leq r$. By Lemma 2, the existence of an $(f,r)$-strategy implies that $\overline{L_r}(GT_c) \leq r$. Thus, it is sufficient to prove that $\overline{L_r}(BT_c) \leq \overline{L_r}(GT_c)$. $BT_c$ is derived from $GT_c$ by adding a leaf to any configuration in $GT_c$ in which *luck* intervenes. By Lemma 1, any addition of such a leaf may only decrease the expected number of rounds. The additional nodes may be ordered by lexicographic order $l_1, l_2, \cdots$. Let $T_0, T_1, T_2, \cdots$, be a (possibly infinite[7]) sequence of

[6] If this probability was $p > 0$ then the expected number of rounds until a configuration in $\mathcal{F}$ is reached would be infinite.

[7] It can be shown that, since *luck* has an $(f,r)$ strategy, the sequence $(T_0, T_1, \cdots)$ must be finite. However, we do not need this fact in our proof.

trees in which $T_0 = GT_c$ and $T_{i+1}$ is obtained by addition of $l_i$ to $T_i$. By the above proposition it holds that $\overline{L_r}(T_i) \geq \overline{L_r}(T_{i+1})$. The lemma follows by observing that the (possibly infinite) sequence $(\overline{L_r}(T_0), \overline{L_r}(T_1), \cdots)$ is a nonincreasing sequence which converges to $\overline{L_r}(BT_c)$. $\Box$

**Theorem 5:** If *luck* has an $(f, r)$-strategy then $\mathcal{PR}$ reaches a configuration in $\mathcal{F}$ within at most $r2^f$ expected number of rounds.

**Proof:** The existence of an $(f, r)$-strategy implies that for every initial configuration $c \in \mathcal{I}$ and for every fair scheduler $S$, an execution that starts with $c$ contains a good block with probability 1. We use Lemma 3 to further show that the expected ordinal number of the first good block in an execution is at most $2^f$. This expected ordinal may only increase if we assume that the probability that the first block in an execution to be a good block is *exactly* $2^{-f}$. In this case, the probability that all the first $i$ blocks in an execution are bad is $(1 - 2^{-f})^i$. Thus, the expected ordinal of the first good block is at most $\sum_{i=1}^{\infty} i \cdot (1 - 2^{-f})^{i-1} \cdot 2^{-f} = 2^f$.

By Lemma 4 and the fact that expectation of a sum is a sum of expectations, the expected number of rounds in an execution until the end of the first good block is at most $2^f \cdot r$. The proof is completed since when reaching the end of the first good block, the system configuration belongs to $\mathcal{F}$. $\Box$

# 4 Self-Stabilizing Leader Election

Self-stabilizing systems were introduced in the seminal paper of Dijkstra, [Dij-74]. In that paper, Dijkstra presents three semi-uniform, self-stabilizing, ring protocols for mutual-exclusion. Other semi-uniform, mutual-exclusion, self-stabilizing ring protocols which work under a stronger adversary, called the *distributed demon* were presented by Brown Gouda and Wu in [BGW-87] and by Burns in [Bu-87]. Two papers considered self-stabilizing, mutual-exclusion protocols for general (connected) graphs: The first was authored by Tchuente, in [Tc-81], who presented a non-uniform protocol for that problem. A semi-uniform protocol for the same problem was presented by Dolev, Israeli and Moran in [DIM-93]. The work of [DIM-93] was the first to propose the read/write atomicity model and their protocol is the first protocol that is self-stabilizing under this model. A self-stabilizing, *id*-based protocol for mutual exclusion in complete graphs is presented by Lamport, in [La-86]. This protocol has exponential time and space complexity. Protocols for leader election in the *id*-based model for a general graph are presented by Arora and Gouda, in [AG-90] and by Afek, Kutten and Yung in [AKY-90].

So far, there are very few uniform self-stabilizing protocols: Burns and Pachl present a uniform, deterministic, self-stabilizing, mutual exclusion protocol for rings of prime size in [BP-89]. Randomized, uniform, self-stabilizing protocols for mutual exclusion in a general graph and for ring orientation are presented by Israeli and Jalfon in [IJ-90] and [IJ-90a] respectively. Afek, Kutten and Yung, in [AKY-93], present once more the protocol which was originally presented in [AKY-90] and prove its correctness. This is an *id*-based leader election protocol for a general graph and they claim that its stabilization time is $O(n^2)$. Then they describe a way to convert their protocol to a uniform one and sketch a correctness proof. A uniform self-stabilizing leader-election protocol for general graphs was obtained independently in [DIM-91].

There are many non self-stabilizing distributed protocols for leader election. We now survey the most related ones: Deterministic, leader-election protocols in *id*-based systems are presented in [Ga-78, Hu-84, KMZ-84, KKM-90]. Uniform, randomized, leader-election protocols are presented in [IR-81, SS-89, MA-89]. Other protocols for uniform systems appear in [AS-88, AA-86].

In this paper, we present uniform, leader-election, self-stabilizing protocols in systems that assume only *read/write atomicity* under which in one atomic step a processor executes either a single read operation or a single write operation. In contrast, most other self-stabilizing protocols assume *composite atomicity* under which a processor may both read and write to the same register in a single atomic step. As we pointed out in [DIM-93], self-stabilizing protocols under read/write atomicity are much harder to design and to analyze. Our protocols work in complete graph systems in which every processor can communicate with every other processor via shared memory. It can be argued that the complete graph topology is too simple: In the *id*-based model there exists a trivial self-stabilizing protocol for this topology in which each processor repeatedly appoints the processor with maximal *id* as a leader. As often happens, it turns out that intricacy of the problem depends on the exact assumptions made on the system. This paper indicates that uniform, self-stabilizing protocols are more subtle even in such simple topologies. In a forthcoming paper, [DIM-94b], we extend the techniques presented here and present uniform, self-stabilizing, leader election protocols for general graph systems. A preliminary report on the results presented in both papers appears in [DIM-91].

## 4.1 Minimum Space Protocol

In this subsection, we present a simple protocol for leader election in which each processor uses a shared one bit register. This protocol is correct in the presence of coarse atomicity that assumes that a coin toss is an internal operation which is not separable from the next **read** or **write** operation. The protocol appears in Figure 1. Each processor communicates with all other processors using a single writer multi reader binary register called the *leader* register, where $leader_i$ denotes the *leader* register of processor $P_i$. Starting the system with any possible combination of binary values of the *leader* registers, the protocol eventually fixes all the *leader* registers but one to hold 0. The single processor whose *leader* value is 1 is the elected leader. The protocol is straight forward: Each processor repeatedly reads all *leader* registers; whenever it sees that no single leader exists, it first decides whether he is a candidate for a leader, and in case it is, it tosses a coin and assigns its value to its register.

```
do forever
    for j := 1 to n (j ≠ i) do leader_i[j] := read(leader_j);
    if (leader_i = 0 and {∀ j ≠ i, | leader_i[j] = 0}) or
       (leader_i = 1 and {∃ j ≠ i, | leader_i[j] = 1}) then
       write leader_i := random({0, 1});
end
```

Figure 1: A minimum space protocol

We define the task $LE$ to be the set of executions in which there exists a single fixed leader throughout the execution.

We define a configuration to be *safe* if it satisfies:

1. For exactly one processor, say $P_i$, $leader_i = 1$ and $\forall j \neq i \ leader_i[j] = 0$.
2. For every other processor, $P_j \neq P_i$, $leader_j = 0$, $leader_j[i] = 1$.

It is easy to verify that in any (fair) execution, $E$, that starts with a safe configuration (as defined above) $P_j$ is a single leader, thus $E \in LE$.

The stabilization time of the protocol is exponential as shown in the next two lemmas:

**Lemma 6:** The protocol stabilizes within at most $2^{O(n)}$ expected number of rounds.

**Proof:** We use Theorem 5 to show that the expected number of rounds before the protocol stabilizes is bounded from above by $2n2^n$. To do this, we present an $(n, 2n)$-strategy for *luck* to win the *sl-game* defined by the protocol, the set of all possible configurations and the set of all safe configurations, respectively. The strategy of *luck* is as follows: Whenever some processor $P_i$ tosses a coin, *luck* intervenes; if for all $j \neq i$, $leader_j = 0$ then *luck* fixes the coin toss to be 1, otherwise it fixes the coin toss to be 0. Since we assume coarse atomicity the protocol implies that at the end of this atomic step $leader_i$ holds the result of the coin toss. The correctness of this strategy follows from the following observations:

- Within less than $2n$ successive rounds, every processor $P_i$ reads all the *leader* registers, and then, if needed, it tosses a coin and writes the outcome in $leader_i$.

- If within the first $2n$ rounds no processor tosses a coin, then the system reaches a safe configuration.

- Under *luck*'s strategy, it holds that after the first coin toss, there exists at least one *leader* register whose value is 1. Moreover, once $leader_j = 1$ for some $j$, there exists a $k$ s.t. $leader_k = 1$ throughout the rest of the execution. To see this, let $S$ be the set of processors whose *leader* register holds 1 after the first coin toss. If there exists a processor $P_k \in S$ which never tosses a coin again, then $leader_k = 1$ forever. Otherwise, every processor in $S$ tosses a coin; in this case, we take $P_k$ to be the last processor in $S$ that tosses a coin. The strategy of *luck* guarantees that during $P_k$'s coin toss all the remaining *leader* values are 0, and hence *luck* sets the result of $P_k$'s coin toss to 1. From now on $leader_k = 1$ and for $j \neq k$, $leader_j = 0$.

- Every processor $P_i$ may toss a coin at most once: If the outcome of $P_i$'s first coin toss is set by *luck* to 0, then in all successive readings $P_i$ finds out that $leader_k = 1$ ($k$ the same as above), and hence it will not toss a coin again. If the outcome of $P_i$'s first coin toss was set to 1 then by the time its coin toss was set to 1, the *leader* values of all other processors are 0. After this atomic step, $P_i$ finds out that it is the only processor whose *leader* value is 1, and hence it will not toss a coin in this case as well (and, in fact, $P_i$ must be $P_k$).

- Within the first $2n$ rounds, the leader value of every processor, except $P_k$, is 0.

Thus, we conclude that after at most $2n$ rounds and within (at most $n$ interventions at most one for each processor) *luck* wins the game. The lemma follows. □

We have presented a simple proof, using the *sl-game* method, that the protocol stabilizes within at most $2n2^n$ rounds.

We conclude this subsection by proving, in the next lemma, that the protocol does not stabilize under fine atomicity in which a coin-toss is a separate atomic step.

**Lemma 7:** The minimum space protocol is not self-stabilizing under fine atomicity.

**Proof:** The following strategy of the scheduler ensures that the protocol never stabilizes under fine atomicity: Start the system in a configuration in which all *leader* registers hold 1. Let one processor notice that it has to toss a coin. If the coin toss result is 1, let this processor toss a coin again until the coin toss result is 0. Now, stop the processor before it writes 0 in its *leader* register, and activate another processor in the same way. Once all processors are about to write 0, let them all write. Now, all the *leader* registers hold 0, and the scheduler can force all processors to write 1 in their registers in a similar way, and so on and so forth. Thus, this strategy ensures that the system never stabilizes. □

## 4.2 A Polynomial Time Protocol

### 4.2.1 The Protocol

In this section, we modify the constant space protocol and obtain a leader-election protocol that reaches a safe configuration within $O(n \log n)$ expected number of rounds and which is correct under fine atomicity. The speed-up in the convergence rate is obtained by augmenting the constant space protocol with a synchronization mechanism. This mechanism guarantees that eventually, between every two successive coin tosses of $P_i$, all the other processors read $leader_i$. The modified protocol appears in Figure 2. To implement the protocol we logically divided the communication register of a processor $P_i$ into the following *fields*: a single binary field $leader_i$, and two ternary fields $arrow(i:j)_i$ and $arrow(j:i)_i$ for every neighbor $P_j$ (i.e. the total number of fields is $1+2(n-1)$). The synchronization mechanism consists of two synchronization subroutines called *synch* and *ack* and a boolean function called *ack_all*. Whenever a processor completes a coin toss, it notifies all other processors by calling *synch*. Subroutine *ack* is called in every pass through the main loop. In this subroutine, the processor reads the *leader* values and acknowledges new coin tosses of other processors. Reading $leader_j$ is executed inside $ack(j)$ to ensure that the acknowledged leader value is the value of the most recent coin toss of $P_j$. The predicate *ack_all* holds when the most recent call to *synch* was acknowledged by all other processors.

We prove the correctness of the algorithm in two stages: First, we show that the synchronization subroutines are self-stabilizing. Then, we use the *sl-game* method to show that the algorithm stabilizes in $O(n \log n)$ rounds.

```
do forever
   write leader_i := coin_i
   if (leader_i = 0 and {∀ j | leader_j.local = 0}) or
      (leader_i = 1 and {∃ j | leader_j.local = 1}) then
      if ack_all then
         coin_i := random({0,1})
         write leader_i := coin_i
         for j := 1 to n (j ≠ i) do synch(j)
      endif
   endif
   for j := 1 to n (j ≠ i) do ack(j)
end
```

Figure 2: The modified Protocol (for $P_i$)

#### 4.2.2 The Synchronization Mechanism

The synchronization mechanism ensures that in every execution, eventually every processor receives acknowledgments from all other processors between every two successive coin tosses. This mechanism uses a data structure called *arrow* which is shared by two processors. Each pair of processors $P_i$ and $P_j$ share *two* arrows: the arrow of $P_i$, denoted by $arrow(i:j)$, and the arrow of $P_j$, denoted by $arrow(j:i)$. $arrow(i:j)$ is implemented by two ternary fields called $arrow(i:j)_i$ and $arrow(i:j)_j$. $arrow(i:j)$ is *directed* from $P_i$ towards $P_j$ when $arrow(i:j)_i \neq arrow(i:j)_j$, otherwise $arrow(i:j)$ is *directed* from $P_j$ to $P_i$. Fields $arrow(i:j)_i$ and $arrow(j:i)_i$ are stored in a single writer single reader atomic register called $r_{ij}$, which is written by $P_i$ and read by $P_j$. Analogously, register $r_{ji}$ is written by $P_j$ and read by $P_i$ and stores the fields $arrow(i:j)_j$ and $arrow(j:i)_j$. The local copy of $arrow(i:j)_i$ ($arrow(i:j)_j$), held by $P_j$ ($P_i$), is denoted by $arrow(i:j)_i.local$ ($arrow(i:j)_j.local$).

After it is stabilized, the synchronization mechanism works as follows: Processor $P_i$ tosses a coin and assigns its value to its leader register and then directs its arrows towards all other processors. Processor $P_j$ reads $leader_i$ as a part of the synchronization mechanism; immediately after reading $leader_i$, $P_j$ redirects $arrow(i:j)$ back to $P_i$ to signal that the new value of $leader_i$ was read. To make sure that the result of the coin toss was read by all other processors, $P_i$ waits until all its arrows are directed back, before it considers whether to toss its coin again. To avoid deadlock, every processor continually reads other processors' arrows (and leader variables) and redirects them whenever needed while it waits for its own arrows to be redirected. The synchronization subroutines appear in Figure 3. The function *ack_all* reads the arrow registers and computes the predicate *ack_all*. Under these definitions *ack_all* holds for $P_i$ if for all $1 \leq j \leq n$, $j \neq i$ $arrow(i:j)_i = arrow(i:j)_j.local$. Procedure *synch(j)* uses the register values read by *ack_all* while $ack(j)$ rereads the arrow register of $P_j$ once more.

**Boolean function** $ack\_all$
$ack\_all := true$
**for** $j := 1$ to $n$ $(j \neq i)$
**do**
  $(arrow(i:j)_j.local, arrow(j:i)_j.local) :=$ **read** $(r_{ji})$
  if $arrow(i:j)_i \neq arrow(i:j)_j.local$ **then** $ack\_all := false$
**endo**

**Procedure** $synch(j)$ (* $arrow(i:j) := (i,j)$ *)
**if** $arrow(i:j)_i = arrow(i:j)_j.local$ **then**
**write** $arrow(i:j)_i := (arrow(i:j)_i + 1)\ mod\ 3$

**Procedure** $ack(j)$ (* $arrow(j:i) := (i,j)$ *)
$(arrow(i:j)_j.local, arrow(j:i)_j.local) :=$ **read** $(r_{ji})$
$leader_j.local :=$ **read** $leader_j$
**write** $arrow(j:i)_i := arrow(j:i)_j.local$

Figure 3: The Synchronization Mechanism (for $P_i$)

Now, we show that the synchronization mechanism, implemented by the arrows is self-stabilizing. Configuration $c$ is said to be *safe* for $arrow(i:j)$ if in any execution that starts from $c$ has the property that:

1. The value of $arrow(i:j)_i$ is changed every time $P_i$ executes $synch(j)$.
2. Between every two successive changes in $arrow(i:j)_i$ $(arrow(i:j)_j)$, there is a change in the value of $arrow(i:j)_j$ $(arrow(i:j)_i$, respectively).

In particular, it means that in every execution starting from $c$, $P_j$ reads $r_{ij}$ and $leader_i$ between every two successive coin tosses of $P_i$. Note that a configuration might be safe for an arrow but not be a safe system configuration, say, because more than a single leader exists.

**Lemma 8:** For every two processors $P_i$ and $P_j$, $P_j$ executes $ack(i)$ every $6n$ rounds. Furthermore, within every $12n$ rounds, every processor completes an entire pass of the main loop.

**Proof:** Execution of all atomic steps in the main loop requires $5n-2$ atomic steps: Execution of line 2 takes one atomic step. Line 4 requires $n-1$ atomic steps. Line 5 and line 6 require one atomic step each, while execution of line 7 and line 8 take $n-1$ and $3n-3$ atomic steps, respectively. Since $ack(i)$ is executed unconditionally in the main loop and since its execution takes 3 atomic steps, it is executed every $5n+1<6n$ rounds. This implies the first claim. The second follows since in any $12n$ successive rounds, the first time that line 2 is executed occurs within the first $6n$ rounds, and within the following $6n$ rounds the complete loop is executed. $\square$

**Lemma 9:**

1. Consider the following equation:

$$arrow(i:j)_i.local = arrow(i:j)_i = arrow(i:j)_j.local = arrow(i:j)_j \qquad (1)$$

   If Equation 1 holds in configuration $c$ for some $i$ and $j$ then $c$ is safe for $arrow(i:j)$.

2. Every execution whose length exceeds $30n$ rounds contains a configuration in which Equation 1 holds.

We bring here the outline of the proof of Lemma 1. A detailed proof appears in [DIM-94a].
Assertion (1) follows by showing that once the system is in a configuration $c$ satisfying Equation 1, either the values of $arrow(i:j)_i.local$, $arrow(i:j)_i$, $arrow(i:j)_j.local$ and $arrow(i:j)_j$ are never changed, or are changed by repeating the following sequence of steps:

1. $P_i$ writes to $r_{ij}$ and changes the value of $arrow(i:j)_i$ ,
2. $P_j$ reads from $r_{ij}$ and assigns $arrow(i:j)_i.local := arrow(i:j)_i$ ,
3. $P_j$ writes to $r_{ji}$ and assigns $arrow(i:j)_j := arrow(i:j)_i.local$ ,
4. $P_i$ reads from $r_{ji}$ and assigns $arrow(i:j)_j.local := arrow(i:j)_j$ ,

and in both cases $arrow(i:j)$ is stabilized in the resulting computations.

To prove assertion (2) of Lemma 1 we first show that if there are $6n$ successive rounds in which $P_i$ does not change $arrow(i:j)_i$, then a configuration in which $arrow(i:j)_i = arrow(i:j)_i.local = arrow(i:j)_j$ is reached; after which, when $P_i$ reads $arrow(i:j)_j$ or is about to write to $arrow(i:j)_i$, a configuration in which equation (1) holds is reached. Then we consider the case in which $P_i$ does change the value of $arrow(i:j)_i$ at least once every $6n$ successive rounds. We show that, in this case, between the third and fourth write operation of $P_i$ it holds that $arrow(i:j)_i = arrow(i:j)_i.local = arrow(i:j)_j$ (this takes $4 \times 6n$ rounds we need additional $6n$ rounds to show that equation (1) holds, which yields the total of $30n$ rounds). The proof of this last part hings on the fact that the register values that implement an arrow are incremented modolu three[8]. The key observation is that the value written in the third write operation of $P_j$ is different from the values of $arrow(i:j)_i.local$, $arrow(i:j)_i$, $arrow(i:j)_j.local$ and $arrow(i:j)_j$ just before this write operation and thus $P_j$ has to read it from $arrow(i:j)_i$ before acknowledging $P_i$ by setting $arrow(i:j)_j$ to the value of $arrow(i:j)_i$.

### 4.2.3 Correctness and Complexity

The set of legitimate executions of this protocol is defined precisely as in the minimum space protocol (the set of executions in which there is exactly one processor with $leader = 1$ and this processor is fixed throughout the execution).

**Definition 5:** A configuration $c$ is *decreasing* if it satisfies properties P1-P3 defined as follows:

**P1** There is at least one processor, say $P_1$, with $leader_1 = coin_1 = 1$.

**P2** $\forall j,k$ $(j \neq k)$, if $leader_j = 1$ then the variable $leader_j.local$ of $P_k$ equals 1 too.

**P3** $\forall j$, if $leader_j = 0$ then $coin_j = 0$.

**Definition 6:** A decreasing configuration with exactly one $j$ such that $leader_j = 1$ is called *critical*.

Informally, in a decreasing configuration, there is at least one processor whose leader value is 1, and for each processor $P$ whose leader value is 1, all other processors know that

[8] In Lemma 3.1 of [DIM-93] is it shown why two values are not sufficient. In [DIM-93] a *close* field is introduced to eliminate this problem. Here we use variables of three values instead, to gain a more symmetric protocol.

$P$'s leader value is 1. Let $c$ be a decreasing configuration. Observe that in $c$, $P_i$ may toss a coin only if $leader_i = 1$. Thus, once a decreasing configuration is reached, the number of processors that hold 1 in their leader variables may only decrease until the system reaches a critical configuration (if at all). We now prove that every execution reaches a safe configuration within expected $O(n \log n)$ rounds. The proof follows the following stages: First we prove that within $O(n)$ expected number of rounds the system reaches a decreasing configuration. Then, we prove that within expected number of $O(n \log n)$ rounds, the system reaches a safe configuration. We begin the proof by showing that if the system does not stabilize then some processor tosses a coin:

**Lemma 10:** Let $E$ be an execution whose length is $19n$ rounds. If the system does not reach a safe configuration during $E$ then at least one processor tosses a coin during $E$.

**Proof:** Assume towards a contradiction that no processor tosses a coin during $E$, and the system does not reach a safe configuration. By this assumption, line 5 is not executed during $E$. The leader value may be changed only in line 6. Since line 5 is not executed during $E$, a processor can change its leader value only during the first round of $E$ (if its first atomic step in $E$ is in line 6). Thus, after the first round of $E$ no leader value is changed. Furthermore, after the first $n$ rounds of $E$, no processor executes $synch(j)$ any more. Let $E_1$ be the subexecution of $18n$ rounds of $E$ in which the leader values are constant and no processor executes $synch(j)$. By Lemma 8, for every $i$ and $j$, $i \neq j$, $P_i$ executes $ack(j)$ during the first $6n$ rounds of $E_1$. Let $c_1$ be the configuration reached after $P_i$ acknowledged $P_j$ for all $i \neq j$. By Lemma 8, every processor completes an execution of the main loop within the next $12n$ rounds of $E_1$. Hence, every processor $P$ executes line 3 after $c_1$. Since the system does not reach a safe configuration, it holds that during $E_1$ either there is more than one processor with $leader = 1$ or the leader value of every processor is 0. In either case, some processor $P_i$ passes the condition of line 3, and executes line 4. Since all processors completed acknowledging each other and since no processor calls $synch$ during $E_1$, $P_i$ also passes the condition in line 4. In its next step, $P_i$ tosses a coin, a contradiction. $\square$

We now prove that the system always reaches a decreasing configuration within an expected number of $O(n)$ rounds.

**Lemma 11:** In every execution, the system reaches a decreasing configuration within $O(n)$ expected number of rounds.

**Proof:** Let $\mathcal{G}$ be the *sl-game* defined by the protocol, by the initial set $\mathcal{I}$ of all possible configurations and by the final set $\mathcal{D}$ of all decreasing configurations. The lemma is proved by the following $(3, 61n)$-strategy for *luck* to win $\mathcal{G}$: *Wait until the system reaches configuration which is safe for all the arrows. Then, wait until the first time (if at all) some processor, P, tosses a coin. Set the first three coin tosses of P to 1.* (If $P$ tosses a coin less than 3 times, then *luck* intervenes less than three times).

Let $E$ be an execution of $\mathcal{G}$ in which *luck* uses its strategy. We now prove that if $E$ does not contain a safe configuration then $E$ reaches a decreasing configuration within at most $61n$ rounds. By Lemma 9, the system reaches a configuration, $c_0$, which is safe for all the arrows within $30n$ rounds. If a safe configuration is not reached within the next $19n$ rounds then by Lemma 10, some processor tosses a coin. Without loss of generality, let this processor be $P_1$. By its strategy, *luck* sets the first three coin tosses of $P_1$ to 1. Due to the synchronization mechanism, the system reaches a configuration $c_1$ in which all other processors have executed $ack(1)$ before $P_1$ tosses a coin for the second time. Note that no processor whose leader value is 0 tosses a coin after finding that $leader_1 = 1$ and executing $ack(1)$, thus both properties (P1) and (P3) hold in $c_1$. These properties continue to hold as long as the leader value of $P_1$ remains 1. Thus, if the system reaches a configuration in which (P2) holds before the leader value of $P_1$ is set to 0 then the system reaches a decreasing configuration.

Let $E_1$ be the maximal sub-execution that starts with $c_1$ during which $leader_1 = 1$. Since the variable $leader_1.local$ of every processor equals 1, no processor with $leader = 0$ assigns 1 in its *leader* during $E_1$. Hence, if for all $i, j$, $P_i$ reads $leader_j$ during $E_1$, then the system reaches a configuration in which (P2) holds. By Lemma 8, every processor $P_i$ executes $ack(j)$ for all $j \neq i$ during any $6n$ successive rounds. Hence, if $E_1$ lasts at least $6n$ rounds then every processor $P_i$ executes $ack(j)$ for all $j \neq i$ and (P2) holds.

Next, we assume that $E_1$ lasts less than $6n$ rounds, and show that every processor $P_i$ executes $ack_j$ for all $i \neq j$. *luck*'s strategy ensures that if $E_1$ lasts less than $6n$ rounds then the second and third coin tosses of $P_1$ occur during $E_1$. Therefore, $P_i$ executes $ack(1)$ twice during $E_1$ (once before each coin toss of $P_1$). Between the first and second execution of $ack(1)$, $P_i$ must execute $ack(j)$ for all other $j$'s.

In both cases, within at most $6n$ rounds following configuration $c_1$, a configuration satisfying property $(P2)$ is reached.

We conclude that the game $\mathcal{G}$ is finished with a decreasing configuration within at most $61n$ rounds: at most $30n$ rounds until the arrows are stabilized, $19n$ rounds until the first coin toss past $c_0$, $6n$ rounds until configuration $c_1$ is reached, and another $6n$ rounds required to reach a decreasing configuration. □

We conclude this section by showing that the system stabilizes within $O(n \log n)$ expected number of rounds. We use the following definition.

**Definition 7:** A configuration $c$ is a *zero* configuration if in $c$, $leader_i = 0$ for all $i$.

**Theorem 12:** In every execution of the protocol, the system reaches a safe configuration within at most $O(n \log n)$ expected number of rounds.

**Proof:** Let $\mathcal{G}_1$ be the *sl-game* like the *sl-game* $\mathcal{G}$ of Lemma 11, only that this time the final set is the set $\mathcal{F}$ of all safe configurations. We extend the strategy for winning $\mathcal{G}$ presented in the proof of Lemma 11 to an *sl*-strategy for $\mathcal{G}_1$, as follows:

*First, use the previous strategy of luck to reach a decreasing configuration. Once a decreasing configuration is reached, continue as follows: Wait until a critical configuration is reached. If the unique processor whose leader variable holds 1 tosses a coin, set the result of the coin toss to be 1.* The theorem is proved by showing that the combined strategy is a $(4, kn \log n) - strategy$ (for some constant $k$) for *luck* to win $\mathcal{G}_1$. For this, it suffices to show that once a decreasing configuration is reached, it takes additional $O(n \log n)$ expected number of rounds to reach a critical configuration (with no interventions of *luck*), and that once a critical configuration is reached, it takes $O(n)$ rounds and one intervention of *luck* to reach a safe configuration.

Every execution that starts with a decreasing execution and reaches a zero configuration after $k$ coin tosses has a subexecution that reaches a critical configuration after at most $k-1$ coin tosses. Hence the expected number of rounds until a critical configuration is reached is bounded from above by the expected number of rounds required from the system to reach a zero configuration which is analyzed as follows:

Let $E$ be an execution that starts with a decreasing configuration whose last configuration is its first zero configuration. By Lemma 8, every processor acknowledge $P_i$ within at most $6n$ rounds, hence $P_i$ executes lines 5 and 6 at least once every $6n$ rounds as long as the condition of line 3 holds. Since all configurations of $E$ except the last one are decreasing, the following holds for every $i$ during $E$: If $leader_i = 1$ then $P_i$ tosses a coin at least once during every $6n$ rounds, and if $leader_i = 0$ then $P_i$ does not toss a coin at all during a zero execution.

Thus, the system reaches a zero configuration once every $P_i$, for which initially $leader_i = 1$, tosses a coin and gets 0 for the first time. For each such $P_i$, the probability that $leader_i = 1$ following $\ell \cdot 6n$ rounds (during which $P_i$ tosses the coin at least $\ell$ times) is at most $(1/2)^\ell$. Hence, the probability to reach a critical configuration following $\ell \cdot 6n$ rounds is greater than $(1-(1/2)^\ell)^n$, and the probability that the system reaches a critical configuration within $6n \cdot 2 \log n$ rounds for $n > 2$ is greater than:

$$(1-(1/2)^{2\log n})^n = (1-(1/(n^2))^n > (1-1/n) > 1/2.$$

Therefore, the expected number of rounds until a critical configuration is reached is smaller than:

$$\Sigma_{i=1}^{\infty} i \cdot 6n \cdot 2 \log n (1/2)^i = 2 \cdot 10 n \log n.$$

Now, we show that if the system reaches a critical configuration then at most one intervention of *luck* (according to its strategy) suffices to bring the system to a safe configuration. Let $P_i$ be the unique processor with $leader_i = 1$ in a given critical configuration $c_r$. Since $c_r$ is decreasing, in any execution $E$ that starts in $c_r$, the first processor to toss a coin (if any) is $P_i$. Thus, If $P_i$ never tosses a coin in $E$ then by Lemma 10, the system reaches a safe configurations within $19n$ additional rounds. Otherwise, $P_i$ tosses a coin and then *luck* sets the result to 1. Let $c_s$ be the configuration that immediately follows this coin toss. We show that $c_s$ is safe, by showing that in any execution that starts from $c_s$, no processor tosses a coin, and hence, in any such execution, it always holds that $leader_i = 1$ and for $j \neq i$, $leader_j = 0$.

In any execution that starts from $c_s$, the first processor to toss a coin cannot be $P_i$, since after its last coin toss, it reads all the leader values, and if none was changed it finds out that it is the unique processor whose leader value is 1, and hence, it does not toss a coin. However, as long as $P_1$ does not change $leader_i$ to 0, the system remains in a decreasing configuration, and hence, no processor $P_j$ for which $leader_j = 0$ will ever toss a coin. We conclude that no processor will ever toss a coin in an execution that starts in $c_s$, as claimed. $\square$

# 5 Conclusions

We presented the *sl-game* method – a novel method for analyzing the time complexity of randomized distributed protocols. The usefulness of the new method is demonstrated by providing realistic bounds on the round complexity of two self-stabilizing leader-election protocols in a complete network. The space complexity of the first protocol is *optimal*, it requires communication bandwidth of a single bit; its round complexity is exponential and it assumes coarse atomicity. The second protocol requires linear space, its round complexity is $O(n \log n)$ and it tolerates fine atomicity.

One of the anonymous referees pointed out that the *sl-game* method can be generalized to the case of a $k$-face dice toss, instead of a coin toss (See also [Do-94]). The reader may verify that when a $k$-face dice result is possible and *luck* has an $(f, r)$-strategy, $\mathcal{PR}$ reaches a configuration in $\mathcal{F}$ within at most $rk^f$ expected number of rounds.

# Acknowledgments

We gladly acknowledge the constructive remarks of four anonymous referees who helped us improving this paper.

## References


[AA-86] K. Abrahamson, A. Adler, L. Higham and D. Kirkpatrick, "Probabilistic solitude verification on a ring", *Proc. of the fifth Annual ACM Symposium on Principles of Distributed Computing*, Quebec City, August 1986, pp. 161-173.

[AS-88] H. Attiya, M. Snir, and M. Warmuth, "Computing on an Anonymous Ring", Journal of the ACM, 35(4), Oct. 1988, pp.845-875.

[Ab-88] K. Abrahamson, "On Achieving Consensus Using a Shared Memory", *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, Toronto Canada, August 1988, pp. 291-302.

[AE-91] E. Angnostou and R. El-Yaniv, "More on the Power of Random Walks: Uniform Self-Stabilizing Algorithms", in *Lecture Notes in Computer Science 579: Distributed Algorithms* (Proceedings of the Fourth International Workshop on Distributed Algorithms, Delphi, Greece, October 1991), S. Toueg, P.G. Spirakis and L. Kirousis, Editors, pp. 31-51, Springer-Verlag, 1992.

[AG-90] A. Arora and M. Gouda: "Distributed Reset", in in *Lecture Notes in Computer Science 472: Foundations of Software Technology and Theoretical Computer Science* (Proceedings of the Tenth Conference on Foundations of Software Technology and Theoretical Computer Science), December 1990, K.V. Nori and C.E. Veni Madhavan (Eds.), pp. 316-331, Springer-Verlag, 1990.

[AKY-90] Y. Afek, S. Kutten and M. Yung, "Memory-Efficient Self-Stabilization on General Networks", in *Lecture Notes in Computer Science 486: Distributed Algorithms* (Proceedings of the Fourth International Workshop on Distributed Algorithms, Bari, Italy, September 1990), J. Van Leeuwen and N. Santoro, Editors, pp. 15-28, Springer-Verlag, 1991.

[AKY-93] Y. Afek, S. Kutten and M. Yung, "Local Detection for Global Self Stabilization", preprint.

[BGW-87] G.M. Brown, M.G. Gouda, and C.L. Wu, "A Self-Stabilizing Token system", *Proc. of the Twentieth Annual Hawaii International Conference on System sciences* 1987, pp. 218-223.

[BP-89] J.E. Burns and J. Pachl, "Uniform Self-Stabilizing Rings", ACM Transactions on Programming Languages and Systems, Vol. 11, No. 2, Apr. 1989, pp. 330-344.

[Bu-87] J.E. Burns, "Self-Stabilizing Rings without Demons", Technical Report GIT-ICS-87/36, Georgia Institute of Technology.

[CC-85] B. Chor, and B. Coan, "A Simple and Efficient randomized Byzantine agreement algorithm," *IEEE Trans. on Software Engineering*, Vol. SE-11, No. 6, pp. 531-539, 1985.

[Dij-74] E.W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control", Communications of the ACM 17,11 1974, pp. 643-644.

[DIM-91] S. Dolev, A. Israeli and S. Moran, "Uniform Dynamic Self-Stabilizing Leader Election", in *Lecture Notes in Computer Science 579: Distributed Algorithms* (Proceedings of the Fifth International Workshop on Distributed Algorithms, Delphi, Greece, October 1991), S. Toueg, P.G. Spirakis and L. Kirousis, Editors, pp. 163-180, Springer-Verlag, 1992.

[DIM-93] 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. Extended abstract appeared in the *Proc. of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, Quebec City, August 1990, pp. 103-118.

[DIM-94a] S. Dolev, A. Israeli and S. Moran, "Uniform Self-Stabilizing Leader Election Part 1: Complete Graph Protocol", Technical Report 94-807, Computer Science Dept., Technion.

[DIM-94b] S. Dolev, A. Israeli and S. Moran, "Uniform Self-Stabilizing Leader Election Part 2: General Graph Protocol", Technical Report 94-039, Department of Computer Science Texas A&M University.

[Do-94] S. Dolev, "Optimal Time Self-Stabilization in Uniform Dynamic Systems," *6th IASTED International Conference on Parallel and Distributed Computing and Systems*, pp. 25-28, 1994.

[DW-93] Dolev, S. and Welch, L. J., "Wait-Free Clock Synchronization," *Proc. of the 12th Annual ACM Symp. on Principles of Distributed Computing*, pp. 97-108, 1993.

[Ga-78] R. G. Gallager, "Finding a leader in networks with $O(E) + O(NlogN)$ messages", Internal Memo., M.I.T., Cambridge, Mass., 1978.

[Hu-84] P. Humblet, "Selecting a leader in a clique in $O(n \log n)$ messages. Inter. Memo., Laboratory for Information and Decision Systems, M.I.T, Cambridge, Mass., 1984.

[IJ-90] A. Israeli and M. Jalfon, "Token Management Schemes and Random walks Yield Self Stabilizing Mutual Exclusion", *Proc. of the Ninth Annual ACM Symposium on Principles of Distributed Computation*, Quebec City, August 1990, pp. 119-132.

[IJ-90a] A. Israeli and M. Jalfon, "Self stabilizing Ring Orientation", in *Lecture Notes in Computer Science 486: Distributed Algorithms* (Proceedings of the Fourth International Workshop on Distributed Algorithms, Bari, Italy, September 1990), J. Van Leeuwen and N. Santoro, Editors, pp. 1-14, Springer-Verlag, 1991.

[IR-81] A. Itai and M. Rodeh, "Probabilistic Methods for Breaking Symmetry in Distributed Networks", Information and Computation, 88(1), 60-87, (1990).

[KKM-90] E. Korach, S. Kutten and S. Moran, "A Modular Technique for the Design of Efficient Distributed Leader Finding Algorithms", *ACM Transactions on Programming Languages and Systems 12*, 1 (1990), pp. 84-101.

[KMZ-84] E. Korach, S. Moran and S.Zaks, "Tight lower and upper bounds for some distributed algorithms for complete network of processors", *Proc. of the 3rd Annual ACM Symposium on Principles od Distributed Computing* (1984), pp. 199-207.

[KP-89] S. Katz and K. J. Perry, "Self-stabilizing extensions for message-passing systems", *Proc. of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, Quebec City, August 1990, pp. 91-101.

[La-86] L. Lamport, "The Mutual Exclusion Problem: Part II - Statement and Solutions", *Journal of the Association for Computing Machinery* , Vol. 33 No. 2 (1986), pp. 327-348.

[MA-89] Y. Matias and Y. Afek, Simple and Efficient Election Algorithms for Anonymous Networks, 1989, manuscript.

[SS-89] B. Schieber and M. Snir "Calling Names on Nameless Networks", *Proceedings of the Eights Annual ACM Symposium on Principles of Distributed Computing*, Edmonton, August 1989, pp. 319-328.

[Tc-81] M. Tchuente, "Sur l'auto-stabilisation dans un réseau d'ordinateurs", RAIRO Inf. Theor. 15 (1981), pp. 47-66