Dijkstra’s Self-Stabilizing Algorithm in Unsupportive Environments

Shlomi Dolev\textsuperscript{1*} and Ted Herman\textsuperscript{2**}

\textsuperscript{1} Department of Computer Science, Ben-Gurion University — Israel
dolev@cs.bgu.ac.il

\textsuperscript{2} Department of Computer Science, University of Iowa
herman@cs.uiowa.edu

Abstract. The first self-stabilizing algorithm published by Dijkstra in 1973 assumed the existence of a central daemon, that activates one processor at time to change state as a function of its own state and the state of a neighbor. Subsequent research has reconsidered this algorithm without the assumption of a central daemon, and under different forms of communication, such as the model of link registers. In all of these investigations, one common feature is the atomicity of communication, whether by shared variables or read/write registers. This paper weakens the atomicity assumptions for the communication model, proposing versions of Dijkstra’s algorithm that tolerate various weaker forms of atomicity, including cases of regular and safe registers. The paper also presents an implementation of Dijkstra’s algorithm based on registers that have probabilistically correct behavior, which requires a notion of weak stabilization, where Markov chains are used to evaluate the probability to be in a safe configuration.

1 Introduction

The self-stabilization concept is not tied to particular system settings. Our work considers several new system settings and demonstrates the applicability of the self-stabilization paradigm to these systems. In particular, we investigate systems with regular and safe registers and present modifications of Dijkstra’s first self-stabilizing algorithm\footnote{Dolev’s work was supported by BGU seed grant.} that stabilizes in these systems.

Dijkstra’s presentation of self-stabilization\footnote{Herman’s work is sponsored by NSF award CAREER 97-9953.} relies on communication by reading neighbor states and updating one machine’s state in one atomic operation. The well-known first algorithm of\footnote{Dolev’s work was supported by BGU seed grant.} represents a machine state by a counter value. Subsequently, this fundamental algorithm has been adapted to link register\footnote{Herman’s work is sponsored by NSF award CAREER 97-9953.} and message-passing\footnote{Dolev’s work was supported by BGU seed grant.} models. These adaptations are straightforward, essentially changing only the domain of the counter values and taking care to compare communication variables to the local state variables in the correct manner. The processing of communication variables, be they message
buffers or communication registers, is atomic in these adaptations of [5]. On the other hand, only few papers [17, 20, 14, 3], address non-atomic communication operations in the context of self-stabilization. Lamport initially demonstrated that interprocess communication without explicitly synchronization is possible [15], and formalizations of less-than-atomic communication were subsequently developed in [21, 16]. The register hierarchy and register constructions of [16] inspired and active research area. The register hierarchy (safe, regular, and atomic registers) has many motivations, including implementation cost for shared register operations. Another view of weaker forms of registers (safe or regular, when compared to atomic) is that they are possible “failure modes” atomic registers. Thus, if we can adapt algorithms such as [5] to accommodate weaker communication assumptions, the result will be an algorithm that not only recovers from transient faults but also deals with certain types of functional errors — hence the title of our paper, the so-called “unsupportive environments.” The idea of combining self-stabilization with other forms of fault-tolerance has previously been studied [1, 12, 9, 10, 2].

We summarize our modifications to [5] as follows. The solution for the regular registers case uses a special label in between writes of labels. In the case of safe registers we prove impossibility results, for the cases in which neighboring processors use a single safe register to communicate between themselves — where the register is/isn’t divided to multiple fields. In the positive side, we define a composite safe register that, roughly speaking, ensures reads return at most one corrupted field and design an algorithm for that case. Subsequently, we consider a stronger model where processors can read the value written in their output registers (therefore avoiding extra writes for refreshes). We present two algorithms for the above case, one that uses unary encoding and another that is based on Gray code.

Then we introduce randomized registers that, roughly speaking, return the “correct value” with probability \( p \). It is impossible to ensure closure in such a system, since all reads may return incorrect values. We introduce the notion of weak self-stabilization for such systems. We use Markov chains to compute the ratio between the number of safe configurations and unsafe configurations in an infinite execution.

Markov chains associate each state (system configuration) with a probability to be in this state during an infinite execution. The fixed probability of the state is a “stabilizing” value. It is clear that the probability is either zero or one in the first configuration. Given the probability of transitions between configurations, one can compute the stable probability in an infinite execution, which is typically greater than zero and less than one. We found the definition of weak stabilization and the use of Markov chains to be an interesting and promising way for extending the applicability of the self-stabilizing concept.

The remainder of the paper is organized as follows. The next section reviews Dijkstra’s algorithm and discusses the problem of adapting it to different register models. Section 3 describes a solution for regular registers. Then in Section 4 we present impossibility results and algorithms for different settings of systems.
that use safe registers. Randomized registers and the use of Markov chains are presented in Section 5. Some concluding remarks are given in Section 6. Detailed proofs are omitted from this extended abstract.

2 Adapting Dijkstra’s Algorithm

For the remainder of the paper, we let \texttt{DijAlg} refer to the first self-stabilizing algorithm of [5]. This algorithm is expressed by guarded assignment statements and based on a ring of \( n \) machines that communicate unidirectionally with atomic (central demon) execution semantics. Generally, conversion of self-stabilizing algorithms from one model to another can be difficult [13]. However the specific case of \texttt{DijAlg} is not difficult to adapt to register models.

For the register models, each “machine” of \texttt{DijAlg} is replaced by a processor, which has an internal state represented by variables and a program counter. Each processor \( p_i \), for \( 0 \leq i < n \), can read from one or more input registers and write to one or more output registers (these communication registers are often called link registers). The unidirectional nature of the ring implies that the set of output registers for \( p_i \) can only be written by \( p_i \) and that the input registers of \( p_{i+1} \) (processor subscripting implicitly uses mod \( n \) arithmetic) are precisely the output registers of \( p_i \). One variant of the register model allows \( p_i \) to write its output registers, but prohibits \( p_i \) from reading its output registers. In the literature on registers, this type of link register is called 1W1R, because the register has exactly one writer \( p_i \) and exactly one reader \( p_{i+1} \). We also consider the case of 1W2R link registers, which allows both \( p_i \) and \( p_{i+1} \) to read the output register of \( p_i \).

The control structure of each processor consists of an infinite iteration of a fixed list of steps that read input registers, perform local calculations and update processor variables, and write to output registers. We call each such iteration of a processor a cycle.

The processor-register model of a system does not specify algorithms using the guarded assignment notation of [5]. Instead, processors are non-terminating automata that continuously take steps in any execution, where an execution is a fair interleaving of processor steps [8]. This means that the direct association of “privilege” with “enabled guard” in [5] does not apply to the processor-register model. Instead, we use the idea of a token, introduced by [4], since each processor is always enabled to take steps.

We do not present a formal statement of what it means to adapt \texttt{DijAlg} to other models in this extended abstract. Informally, an adaptation of \texttt{DijAlg} satisfies the following constraints for each processor: except for the program counter and temporary variables used to read communication registers or used for local calculations, there is one counter variable that represents the state of a machine of \texttt{DijAlg}; processor \( p_0 \) plays the role of the exceptional machine of \texttt{DijAlg} by incrementing its counter (modulo some value \( K \)) when the values it reads from its input registers represent its current counter value, and otherwise its counter value does not change; and processor \( p_i, i \neq 0 \), plays the role of
an unexceptional machine of DijAlg, changing its counter only if it reads input registers that represent a value different from its current counter, and then \( p_i \) assigns its counter to be equal to the representation from the input registers. A “token” is thus equivalent to the conditions that enable a processor to change its counter.

In subsequent sections we investigate adaptations of DijAlg using non-atomic registers. Observe that, since the nature of DijAlg's legitimate behavior is single-token circulation in the ring (mutual exclusion), it follows that transferring a token from one processor to the next is essentially atomic — the token only moves in one direction and cannot reverse course. Thus the challenge of using non-atomic registers is to simulate this atomic behavior. Section 5 weakens this atomicity for a model of registers that are only probabilistically correct.

3 Regular Registers

Before we introduce our results for the case of regular registers let us present “folklore” results concerning read/write registers.

Read/Write Atomicity: It is known that \( n - 1 \) labels are sufficient for the convergence of DijAlg assuming a central daemon, where \( n \) is the number of processors in the ring. We next prove that \( n - 2 \) labels are not sufficient.

Lower bound: Consider the case of \( n - 2 \) states in a system of \( n = 5 \) processors. Thus there are three possible processor states, which we label \{0,1,2\}. To prove impossibility we demonstrate a non-converging sequence of transitions (the key to constructing the sequence is to maintain all three types of labels in each system state, which violates the key assumption for the proof of convergence).

\[
\{0,0,2,1,0\} \xrightarrow{p_1} \{1,0,2,1,0\} \xrightarrow{p_2} \{1,0,2,1,1\} \\
\xrightarrow{p_3} \{1,0,2,2,1\} \xrightarrow{p_4} \{1,0,0,2,1\} \xrightarrow{p_5} \{1,1,0,2,1\}
\]

We now present a reduction (see [8]) of a ring with \( 2n \) processors that is activated by a central daemon to a ring with \( n \) processors that assumes read write atomicity. We conclude that at least \( 2n - 1 \) states are required.

Each processor \( p_j \) has an internal variable in which \( p_j \) stores the value \( p_j \) reads from \( p_{j-1} \). Each read is a copy to an internal variable and each write is a copy of internal variable to a register. Thus, we have in fact a ring of \( 2n \) processors in a system with a central daemon. Hence, \( 2n - 1 \) states are required and are sufficient.

Regular registers: We now turn to the design of an algorithm for the case of regular registers. Informally, a regular register has the property that a read operation concurrent with a write operation can return either the “old” or “new” value. More formally, to define a regular register \( r \) we need to define the possible values that a read operation from \( r \) returns. Let \( x^0 \) be the value of the last write operation to \( r \) that ends prior to the beginning of the read operation (let \( x^0 \) be the initial value of \( r \) if no such write exists).
A read operation from a regular register \( r \) that is not executed concurrently with a write operation to \( r \) returns \( x^0 \). A read operation from a regular register \( r \) that is executed concurrently with a write of a value \( x^1 \) returns either \( x^0 \) or \( x^1 \). Note that more generally, a read concurrent with a sequence of write operations of the values \( x^1, x^2, \ldots, x^m \) to \( r \) could return any \( x^k \), \( 0 \leq k \leq m \), however once a read returns \( x^k \) for \( k > 0 \), no subsequent read by the same reader will return \( x^j \) for \( j < k - 1 \) (for two successive read operations, there is at most one write operation concurrent with both).

\[
\begin{array}{c|c|c|}
 s_1 & s_2 & s_3 \\
0 & 0 & 0 & p_1 \text{ starts to write 1} \\
1 & 0 & 0 & p_1 \text{ still writing 1, and } p_2 \text{ reads 1} \\
1 & 1 & 0 & p_1 \text{ still writing 1, and } p_2 \text{ writes 1} \\
1 & 1 & 0 & p_1 \text{ still writing 1, } p_2 \text{ writes 1, and } p_3 \text{ reads 1} \\
1 & 1 & 0 & p_1 \text{ still writing 1, } p_2 \text{ reads 0} \\
1 & 0 & 1 & p_2 \text{ writes 0, and } p_3 \text{ writes 1} \\
1 & 0 & 1 & p_1 \text{ reads 1, } p_2 \text{ reads 1, and } p_3 \text{ reads 0} \\
\end{array}
\]

Fig. 1. Straightforward regular register implementation fails

A naïve implementation of DijAlg using regular registers may result in the execution presented in Figure 1. We have started in a safe configuration in which all the values (in the registers and the internal variables) are 0 and we have reached a configuration in which all the processors may simultaneously change a state.

To overcome the above difficulty we introduce a new value \( \perp \) that is written before any change of a value of a register (the domain of register values thus includes all counter values and the new value \( \perp \)). The algorithm for the case of regular registers appears in Figure 2. In the figure, \( IR_i \) is the input register for \( p_i \) (thus \( IR_p \) is the output register of \( p_{i-1} \)). Variable \( x_i \) contains the counter from DijAlg, and variable \( t_i \) is introduced to emphasize the fine-grained atomicity of the model (one step reads a register, and the value it returns is tested in another step). If processor \( p_i \) reads a value \( \perp \) from an input register, it ignores the value (see line 10 of the protocol).

A safe configuration is a configuration in which all the registers have the same value, say \( x \), and every read operation that has already started will return \( x \). For simplicity we assume there are \( 2n + 1 \) states. Therefore, it is clear that a state is missing in the initial configuration, say the state \( y \). Hence, when \( p_1 \) writes \( y \), \( p_1 \) does not change its state before reading it from \( p_n \). \( p_n \) can read \( y \) only when \( p_{n-1} \) has the state \( y \). Any read operation of \( p_{n-1} \) that starts following the write operation that assigns \( y \) to \( p_{n-1} \) may return either \( \perp \) or \( y \), which is effectively \( y \) (see lines 3 to 6 and 10 to 13 of the code).
4 Safe Registers

Safe registers have the weakest properties of any in Lamport’s hierarchy. A read concurrent with a write to a safe register can return any value in the register’s domain, even if the value being written is already equal to what the register contains. There are two cases to consider for the model of safe registers. If a processor is unable to read the register(s) that it writes, we can show that DijAlg cannot be implemented. We initially consider the model of a single link register for each processor under the restriction that a writer is unable to read its output registers, that is, the model of 1W1R registers.

The construction of a regular 1W1R boolean register from a safe boolean 1W1R register given in [16] is simple: the writer skips actually writing to the register if the new value is the same as the value already in the register. This is possible because the writer keeps an internal variable equal to the current value of the register. However, this technique is not self-stabilizing because the initial value of the writer’s internal variable may not be correct.

**Lemma 1.** DijAlg cannot be adapted by using only a single 1W1R safe register between $p_i$ and $p_{i+1}$.

*Proof.* Processor $p_i$ ($i \neq 1$) that copies from the output register of $p_{i-1}$ must continually rewrite its output register for $p_{i+1}$ — otherwise there can be a deadlock where the value written by $p_i$ is different from the value $p_i$ reads from $p_{i-1}$. Similarly, $p_i$ must repeatedly write, otherwise there can be a deadlock where the value written by $p_i$ is the same as the value $p_i$ reads from $p_{i-1}$. Therefore, processors continually write into their output registers. Since all processors repeatedly write their output registers, we can construct an execution where reads
Multiple fields safe register: Lemma 1 can be generalized, and we sketch the argument here. We consider the case of multiple safe registers per processor, but where processors cannot read the registers they write. Suppose each processor $p_i$ has $m$ safe registers to write, which $p_{i+1}$ reads, and also $p_i$ reads $m$ safe registers written by $p_{i-1}$. If a protocol allows a state in which a processor does not write any of its registers so long as its state does not change, then we may construct a deadlock because the local state of the processor differs from the encoding of values contained in its output registers. Therefore, in any implementation of the protocol, we can construct an execution fragment so that any chosen processor $p_i$ writes at least some of its registers $t$ times, for arbitrary $t > 0$, and during the same execution fragment, $p_{i-1}$ takes no steps. Moreover, if $p_i$ does not write to all $m$ registers, then the registers it does not write can have arbitrary values inherited from the initial state. Therefore, $p_{i+1}$ can read any value from $p_i$, since at each step of $p_{i+1}$ reading one of the $m$ registers written by $p_i$, we can construct an execution in which $p_i$ is concurrently writing to the same safe register. Because $p_{i+1}$ can read any value, it is possible that for $i \neq n$ that $p_{i+1}$ reads a value equal to its own current value, which for DijAlg, means that $p_{i+1}$ will maintain its current value rather than changing it; for the case $i = n$, there is an execution where each time $p_i$ reads its input registers, the value read differs from its own value, and again $p_i$ makes no change to its current value. These situations can repeat indefinitely with no processor entering the critical section.

Composite safe register: Next we sketch a solution in which fields of the registers can be written and the entire register is read at once. We call such a register composite safe register. A read from a composite safe register may return an arbitrary value for at most one of the register fields, a field in which a write is executed concurrently to the read\(^1\). We note that there is a natural extension of our algorithm in which at most $k$ fields of a register may return an arbitrary value.

Each bit of the label value is stored in three 1-bit safe registers (three fields). This will ensure that a read during a refresh operation will return the value of the register. Assume that the value 101 is stored in nine 1-bit safe registers as 111000111. Assume further that a processor refreshes the value written in these registers each time writing in one of the 1-bit safe registers. A read operation returns the value of the entire composite safe register in which at most one bit is wrong. The Hamming distance ensures that the original value of the label bit can be determined.

To allow a value change we add a three bits guard value. Hence, the composite safe register has three bits that function as a guard value and $3 \times 2(n + 1)$ bits for the label.

\(^1\) This assumption reflects reality in system in which a read operation is much faster than a write operation.
A processor \( p_i, i \neq 1 \), that reads a new value from \( p_{i-1} \) first sets the guard value to 0 (writes 000 in the guard bits), and then changes the value of the label. \( p_i \) writes 111 to its guard bits once \( p_i \) finishes updating the label.

A processor \( p_i \) that reads a guard value 0 does not use the value read. When \( p_i \) reads a guard value 1 it examines the value it read.

The correctness proof starts in convincing ourselves that after the first time a processor \( p_i \) refreshes (or writes a new value in) its register any read operation from its register (that returns a value) results in the last value written to this register. \( p_i \) eventually writes a non-existing label, this label cleans the system. More details are omitted from this extended abstract.

**Safe registers with reads instead of refreshes:** Given the above impossibility results, we examine settings where a processor can read the contents of the registers in which it writes. Consider \( 2n + 1 \) single bit, safe, 1W2R registers rather than a single register per processor. Each processor maintains a counter with domain \([1, 2n + 1]\) for \( \text{DijAlg} \). Unary encoding represents this counter: for a counter value \( k \), the proper encoding is to write all registers 0 except for the register with index \( k \), which has value 1. Figure 3 gives an adaptation of \( \text{DijAlg} \) for this 1W2R model. Lines 2–7 are concerned with correcting the values of registers to agree with internal counter values, but such writing is only done where needed. This correction is for the case of incorrect values in illegitimate initial configurations. However, for convenience, we let lines 2–7 also do the work of transmitting a new counter value (since lines 2–7 are repeated after lines 8–13 in the execution of each processor).

A legitimate configuration for this protocol is that each register vector represents the processor’s last counter value (it differs only when a processor updates its counter) and counters correspond to \( \text{DijAlg} \).

**Lemma 2.** Figure 3 is a self-stabilizing adaptation of \( \text{DijAlg} \).

**Proof.** There are two proof obligations, stability (closure) from legitimate configurations and convergence from arbitrary configurations to legitimate ones.

**Closure.** It is straightforward to verify that in any processor cycle from a legitimate configuration, a processor writes to at most two registers as it changes the counter value. Thus when the neighbor reads these registers, at most two reads can have incorrect values due to concurrent writing. If both have correct values, the token passes correctly (a subsequent read by the process can still obtain an incorrect value, but only by getting 0 for all reads, which causes no harm). If both have incorrect values, then the reader observes no change in counter values. If just one returns an incorrect value, then the reader observes parity of zero, which is harmless. This reasoning shows that the protocol is stable.

**Convergence.** The remaining task is to verify that the protocol guarantees to reach a legitimate configuration in any execution. Suppose all processors have completed at least one cycle of statements 1-13. In the subsequent execution, a processor only writes a register if that register requires change to agree with the processor’s counter. Note that by standard arguments, no deadlock is possible in this system and that \( p_1 \) increments its counter infinitely many times in an
Fig. 3.
DijAlg adaptation for Safe Registers

evaluation. It is still possible that one processor can read more than two incorrect values due to concurrent writes (consider an initial state with many counter values; as these values are propagated to some \(p_i\), it could be that \(p_i\) happens to read many registers concurrent with \(p_i\) writing to them). Since the counter range is \([1, 2n + 1]\) and there are \(n\) processors, it follows that at least one counter value \(t\) is not present in the system. By the arguments given for the proof of closure, no processor incorrectly reads input registers to get the value \(t\) in such a configuration. Because \(p_i\) increments \(x_1\) infinitely, we can suppose \(x_1 = t\) but no other processor or register encoding equals \(t\), and by standard arguments (and the propagation of values observed in the proof of closure), a legitimate configuration eventually is reached.

A standard construction of 1W2R registers from 1W1R registers is just to allocate an added new output register and arrange for the writer to ensure the outputs are duplicates. The reader may wonder if such a construction contradicts Lemma 1 and its generalization. There are two interesting cases to examine. First, we reject any protocol where \(p_{i+1}\) has a new output register that \(p_i\) reads, since such an adaptation of DijAlg would violate the unidirectional model constraint. Second, adding a new 1W1R register written by \(p_i\) and read only by \(p_i\) would be equivalent to an internal variable, which we have already considered in the proof of Lemma 1. Thus, under our constraints on adaptation of DijAlg, the models of 1W1R and 1W2R registers differ in capability.
The protocol of Figure 3 uses an expensive encoding of counter values, requiring $2n + 1$ separate registers. The argument for closure shows that changing a counter and transmitting it is effectively an atomic transfer of the value — once the new value is observed, then any subsequent read of the registers either returns the new value or some invalid value (where the sum of bits does not equal 1), which is ignored. Note that this technique is not an implementation of an atomic register from safe registers; it is specific to the implementation of DijAlg.

Can we do better than using $2n + 1$ registers? The following protocol uses the Gray code representation [11] of the counter, plus a extra bit for parity. The number of registers per processor is $m + 1$ where $m = \lceil \lg (2n + 1) \rceil$.

\begin{verbatim}
1  $p_1$:  do forever
2    do $k := 1$ to $m$
3       read $t_1 := |R_2$
4       if $t_1 \neq \text{Graycode}(x_1)[k]$
5          write $|R_3[k] := \text{Graycode}(x_1)[k]$
6       read $t_1 := |R_2[m + 1]$
7       if $t_1 \neq \text{parity}(|\text{Graycode}(x_1))$
8          write $|R_2[m + 1] := \text{parity}(|\text{Graycode}(x_1))$
9    do $k := 1$ to $m$
10       read $g_1[k] := |R_1[k]$
11       read $t_1 := |R_1[m + 1]$
12       if $t_1 = \text{parity}(g_1) \wedge \text{Graycode}(x_1) = g_1$
13          $x_1 := (x_1 + 1) \mod (2n + 1)$
\end{verbatim}

\begin{verbatim}
1  $p_i$ (i \neq 1):  do forever
2-11 (similar to code for $p_1$)
12       if $t_i = \text{parity}(g_i) \wedge \text{Graycode}(x_i) \neq g_i$
13          $x_i := g_i$
\end{verbatim}

\textbf{Fig. 4.}
\textit{DijAlg using Gray Code}

\textbf{Lemma 3.} \textit{Figure 4 is a self-stabilizing adaptation of DijAlg.}

\textit{Proof.} The closure argument is the same as given in the proof of Lemma 2, inspecting each of the four cases of reading overlapping with writing of the two bits that change when a processor changes its counter and writes the one new Gray code bit and the parity bit. In each case, the neighbor processor either reads the old value, or ignores the values it reads (because parity is incorrect), or obtains the new counter value. The change from old to new counter value is essentially atomic.
Proof of convergence requires new arguments. Consider some configuration of an execution prior to which each processor has completed at least two cycles of statements 1-13 in Figure 4, so that output registers agree with counter values (unless the processor has read a new value and updated its counter). Observe that thereafter, if processor $p_i$ successively reads two different Gray code values from its input registers, each with correct parity, then $p_{i-1}$ concurrently wrote at least once to its output registers. Moreover, if $p_i$ successively reads $k$ different Gray code values with correct parity, then $p_{i-1}$ wrote at least $k - 1$ times a new counter value and read at least $k - 1$ times from its own input registers (in turn, written by $p_{i-2}$). A consequence of these observations is that if $p_i$ successively reads $k$ different counter values with correct parity, then $p_{n-k}$ wrote at least one new counter value in the same period. In particular, if $p_i$ successively reads $n + 2$ different counter values, then we may assert that $p_n$ read $p_1$’s output registers and wrote a new counter in the same period. By the standard argument refuting deadlock, processor $p_1$ increments its counter infinitely often in any execution. Therefore we can consider an execution suffix starting with $x_1 = 0$. In the reflected Gray code [11], the high-order bit starting from $x_1 = 0$ does not change until the counter has incremented $2^n$ times. Therefore, until $p_1$ has incremented $x_1$ at least $2^n$ times, any read by $p_2$ obtains a value with zero in the high-order bit. The observations above imply that, before $x_1$ changes at the high-order bit, each processor has copied some counter value that originated via $p_1$ — such counter values may be inaccurate due to reads overlapping writes or more than one write (bit change) for one scan of a set of registers, however the value for the high-order bit stabilizes to zero in this execution fragment. In a configuration where no counter or register set has 1 in the high-order bit, the event of $p_1$ changing the high-order bit creates a unique occurrence of 1 in that position. Since $p_1$ does not again change its counter until observing the same value from $p_n$, convergence is guaranteed.

5 Randomized State Reads and Weak Stabilization

Consider a system with a probabilistic central daemon, in any given configuration the daemon activates each of the processors with equal probability. A system is weakly stabilizing if, in any execution, the probability that the system remains in any set of illegitimate configurations is zero. This definition implies that a weakly stabilizing system has the property that its state is infinitely often legitimate. In addition, one can sum up the probabilities for being in a legitimate state and use this value to compare algorithms.

To apply the definition of weak stabilization, we model register behavior probabilistically: a processor that makes a transition may “read” an incorrect value and therefore make an errant transition. When the daemon selects a processor, we consider the transition by that processor to be one full cycle of reading its input and writing its output register atomically. We use Markov chains to analyze the stationary probabilities, i.e., to determine the probability that the
system will be in a legitimate configuration. See [18] for a description of Markov chains.

![Transition Probabilities](image)

**Fig. 5.** Transition Probabilities (factorized by 3)

We continue describing our approach using a system of three processors and two states. A read of a neighboring state returns with probability $p$ the correct value, and with probability $1 - p$ the complement of the correct value. There are eight states for this system: Fig. 5 shows the transition diagram for this system. We explain the figure by considering the configuration 111. With probability $1/3$, the daemon activates $p_1$, and $p_1$ reads with probability $1 - p$ the incorrect value 0, which is ignored by DijAlg; thus the joint probability for this transition from 111
to 111 is $(1-p)/3$. Two other transitions from 111 to 111 are also possible: either $p_1$ or $p_2$ could read correctly with probability $p$, thus the probability for each of these transitions is $p/3$. The sum of all cases for this transition is therefore $(1-p)/3 + 2p/3 = (1 + p)/3$. To simplify the presentation, all transition arcs have an implicit factor of 3, so the 111 to 111 arc in Fig. 5 is $(1 + p)$. Similar case analysis derives probabilities for the other arcs shown in the figure. Each configuration has four outgoing arrows, one arrow for each state change of a processor, and one for staying in the same state.

We now choose specific values for $p$ and compute powers of the transition probability matrix $P$, such that the matrix in power $i$ and $i + 1$ are equal ($P^i = P^{i+1}$). Each entry $P_{jk}$ of the $8 \times 8$ transition probability matrix $P$ contains the probability for a transition from configuration $j$ to configuration $k$. The equilibrium probability of being in a legal configuration (i.e., not in the configurations 010 or 101) is then derived from $P^i$. The following table shows different values for $p$ (1, 3/4, 1/2, 1/4) and the corresponding equilibrium vector $e^T$; two figures display the transition matrix $P$ for the cases of $p = 1$ and $p = 3/4$.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Equilibrium Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fig 6</td>
<td>1/6.1/6.1/6.1/6.1/6.1/6.1/6.1/6</td>
</tr>
<tr>
<td></td>
<td>1/4/1/12.1/12.1/12.1/12.1/12.1/12.1/12</td>
</tr>
</tbody>
</table>

The vectors show that the equilibrium probability for illegitimate configurations is zero for the deterministic case, then increasing as $p$ reduces. Clearly, we can investigate the behavior of other systems with a range of probabilities, using the same approach. The results can assist us in comparing different system designs.

**Lemma 4.** The adaptation of DiAlg is weakly stabilizing when register reads are correct with probability $p > 0$.

### 6 Conclusion

This paper presents a number of results related to weakening the model of atomic link registers for the unidirectional ring model associated with DiAlg. Our results are both positive (the constructions for regular and safe registers) and negative (the impossibility for 1W1R safe registers). Some similar experiences with the difficulties of self-stabilizing register constructions are reported in [14], however the problem adapting DiAlg has additional constraints and also advantages: the constraint of unidirectional communication rules out certain techniques, but the ring topology does provide sufficient feedback (eventually information flows from $p_i$ back to $p_{i-1}$) to make constructions possible.
The introduction of probabilistic registers motivates a weakened form of self-stabilization. (Another model of probabilistically correct registers appears in [19].) Our model of probabilistically correct registers could be overly pessimistic: it could be interesting to refine the model so that reads are always correct if there is no concurrent write, but only correct with probability $p$ when a read overlaps a write. The weakly stabilizing protocol does not guarantee closure, since there is always the possibility of a read returning an incorrect value. The usefulness of this kind of weakening of stability is typically associated with control theory, e.g., in [22] the goal is to find systems such that all trajectories visit the “good” states infinitely often. Though rarely formalized in the literature of self-stabilization, this reasoning is implicit in many papers: each new fault perturbs the system, and provided the execution is fault-free for long enough, the system returns to a legitimate state. Our probabilistic model quantifies both fault and scheduler probabilities to characterize executions with a Markov model.

**Acknowledgement.** We are grateful to the reviewers, whose comments and suggestions helped us to improve the presentation of this paper.
References
