Energy optimisation in resilient self-stabilizing processes (extended abstract)

Adrian Kosowski and Łukasz Kuszner
Gdańsk University of Technology, Poland
Department of Algorithms and System Modeling

E-mail: {kosowski,kuszner}@sphere.pl

Abstract

When performing an algorithm in the self-stabilizing model, a distributed system must achieve a desirable global state regardless of the initial state, whereas each node has only local information about the system. Depending on adopted assumptions concerning the model of simultaneous execution and scheduler fairness, some algorithms may differ in stabilization time or possibly not stabilize at all. Surprisingly, we show that the class of polynomially-solvable self-stabilizing problems is invariant with respect to the assumption of weak scheduler fairness. Furthermore, for systems with a single distinguished vertex we prove a much stronger equivalence, stating that synchronisation, the existence of a central scheduler and its fairness have no influence on polynomial stabilization time.

Key words: self-stabilization, asynchronous system, polynomial-time complexity, distributed algorithms.

1 Introduction

The concept of self-stabilization

In all considerations it is assumed that the distributed system consists of nodes connected by communication channels. Each node maintains variables which determine its *local state*. The *configuration* of the system is the union of all local states of its nodes. This model is regarded as a good abstraction for real objects such as peer-to-peer networks.

The system ought to be constructed in such a way as to guarantee that it works correctly, i.e. persists in a legitimate configuration, even though some perturbations can bring it to an illegitimate state. It is desirable that it returns to a legitimate state without any external intervention. Self-stabilization, a concept introduced by Dijkstra [3] in 1974, can be thought of as a technique for designing such resilient systems. A *self-stabilizing system* is one which is able to achieve a legitimate configuration starting from any possible configuration.

The time complexity of such algorithms is expressed either in terms of the number of actions performed by nodes of the system, or in terms of the number of asynchronous rounds before system stabilization. Here we assume the former model which is more suitable for the energy cost of executions. Many asynchronous self-stabilizing algorithms have been proven to operate in polynomial time when the system is controled by a central daemon (e.g. [5, 6, 7, 8]), or when time complexity is measured in asynchronous rounds (see the book by Dolev [4] for detailed bibliography). On the other hand, algorithms guaranteed to stabilize in a polynomial number of moves have been scarce in the more restrictive model of a system controled by a distributed daemon with no fairness assumptions, i.e. with nodes moving independently of each other in arbitrary order.

2 Computational models for self-stabilizing systems

A distributed system can be modeled by a connected graph $G = (V, E)$, where vertex set V corresponds to system nodes and the set of edges E denotes communication links between them. The algorithm for each vertex v is given as a sequence of rules $R_1, R_2, \ldots R_k$, where each rule R_i is of the form: $\textbf{if}\, P_{R_i}(v)\, \textbf{then}\, A_{R_i}$, where $P_{R_i}(v)$ is a predicate over local states of v and its neighbors, and A_{R_i} is an action changing a local state of v. We say that v is *active* if at least one of its predicates is true. A *move* of v according to algorithm $\mathcal{A}$ proceeds as follows: first predicate $P_{R_1}(v)$ is evaluated and if it is true, action A_{R_1} is taken; then subsequent rules are evaluated.

By a *computational step* (or step) we mean a pair of configurations (c_i, c_j), such that c_j can be reached from c_i by a parallel move taken by some subset of vertices. An *execution* $e = ((c_1, c_2), (c_2, c_3) \ldots)$ is a sequence of computational steps.

Schedulers

We assume the existence of a scheduler (or daemon), which selects from among the nodes a subset to perform the next step of an algorithm. Generally two types of such daemons are considered: a central daemon and distributed daemon. The former selects at most one vertex at a time (or — equivalently from the point of view of computation — an independent set of vertices) to execute its actions, while the latter is capable of selecting an arbitrary set of active vertices. A possible approach to conversion between these two types of daemons has been shown in [1].

Fairness assumptions

A *fair execution* is an execution in which every vertex which is active infinitely often performs an infinite number of moves. A *weakly fair execution* is an execution in which there is no suffix such that there exists a vertex which is persistently active and performs no moves.

Legal behaviour

In order to define legal behavior of a system, a subset of *legal configurations* must be distinguished for a given problem, from amongst the set of all configurations. We say that algorithm $\mathcal{A}$ solves a problem, identified by its set of legal configurations, if every system execution e under algorithm $\mathcal{A}$ has a legal suffix, that is there exists a configuration c in execution e such that all possible consequent configurations are legal.

Complexity measure

By the *complexity* of an execution we mean the number of configurations in a sequence till the moment a *safe* configuration has been reached. A configuration c is safe if for every execution starting in c all possible consequent configurations are legal.

3 Algorithm transformations

First, we will provide a convenient notation which will allow us to present various self-stabilizing algorithm transformers (consult [2] and its references).

Suppose that $\mathcal{A}$ is a self-stabilizing algorithm consisting of k rules R_i, for $1 \leq i \leq k$. By a *transformer* we will mean an algorithm $\mathcal{T}_A$ consisting of a set of rules T_i, for $1 \leq i \leq l$. It is assumed that state variables can appear in one set of rules only — that of algorithm $\mathcal{A}$ or that of algorithm $\mathcal{T}_A$. Now, we can construct a new algorithm $\mathcal{A}' = \mathcal{T}_A \circ \mathcal{A}$ consisting of $l + k + 1$ rules, in the following order: $T_1, T_2, \ldots T_l, R'_1, R'_2, \ldots R'_k, R_{\text{off}}$, where rule R'_i has the form:

```
$R'_i$:  if Predicate($R_i$) $\wedge A(v)$
         then Action($R_i$)
```

while rule R_{off} is defined as:

```
$R_{\text{off}}$:  if $A(v)$
         then $A(v) := false$
```

Informally, the distinguished local variable A is used as a switch to enable rules of algorithm $\mathcal{A}$. If the value of switch A is set for vertex v during a computational step, v is said to be *enabled* with respect to algorithm $\mathcal{A}$ in this step (note that the value of A will always be unset directly before the end of the step).

4 Ensuring weak fairness under a central daemon

In the first approach, we will provide a solution to ensure weakly fair executions. Suppose that we are given an algorithm $\mathcal{A}$ which is self-stabilizing under the assumption of weak fairness of the system. Now, we will construct a transformer $\mathcal{T}_A$ to ensure the correct behaviour of $\mathcal{T}_A \circ \mathcal{A}$ in a system where weak fairness is not guaranteed.

Algorithm 2: **UNBOUNDED WEAK FAIRNESS UNDER CENTRAL DAEMON**

```
$T_1$:  if $\exists_{u \in N(v)} c(u) = c(v)$
        then $c(v) := 1 + \max\{c(u) \mid u \in N(v)\}$

$T_2$:  if $c(v) > \max\{0, 1 + c(u) \mid u \in N(v) \wedge$
          $c(u) < c(v)\}$
        then $c(v) := \max\{0, 1 + c(u) \mid u \in N(v) \wedge$
            $c(u) < c(v)\}$

$T_3$:  if $c(v) = 0$
        then $c(v) := 1 + \max\{c(u) \mid u \in N(v)\}$;
            $A(v) := true$
```

Intuitively, values stored in local variables c can be seen as a coloring of the system graph. Such a coloring implies local priority queues, defined in such a way that a node with a smaller color has higher priority. Nodes colored with color 0 are allowed to switch on the rules of the underlying algorithm $\mathcal{A}$. Rule T_1 prevents illegal situation where two neighboring nodes have the same color, rule T_2 ensures that there are no gaps in code sequences, while rule T_3 enables the switch. The latter statement can be formulated as follows.

Corollary 1 *In a computational step of algorithm $\mathcal{T}_A \circ \mathcal{A}$, vertex v is enabled with respect to algorithm $\mathcal{A}$ iff v performs rule T_3 in this step.*

Lemma 1 *In any execution e of algorithm $\mathcal{T}_A \circ \mathcal{A}$, any sequence of consecutive steps involving only rules T_1 or T_2 is shorter than $2n^2$ steps.*

Lemma 2 *The number of moves using rule T_3 in any execution e of algorithm $\mathcal{T}_A \circ \mathcal{A}$ is infinite; moreover, any sequence of at least $2n^2$ consecutive steps involves rule T_3 at least once.*

Lemma 3 *The number of moves performed by any two neigboring vertex u and v according to rule T_3 in each prefix of some execution e of algorithm $\mathcal{T}_A \circ \mathcal{A}$ differs by at most one.*

Theorem 4 *If algorithm $\mathcal{A}$ stabilizes under a central daemon in $f(n)$ moves, then algorithm $\mathcal{T}_A \circ \mathcal{A}$ reaches a safe configuration in $O(n^3 \operatorname{diam}(G))f(n)$ steps.*

4.1 Halting property

In the previous section we have proved that Algorithm 2 is able to transform a self-stabilizing algorithm working under central daemon assuming weak fairness of executions into one without this assumption. Unfortunately, the transformed algorithm always runs infinitely and never stops, i.e. it performs moves even if the algorithm $\mathcal{A}$ which underwent transformation has completed its goal and all the vertices are inactive according to its rules (and in a safe configuration). We now give an algorithm which is capable of halting.

To achieve this, we add a variable s, which can be interpreted as the 'scent' of a node with at least one rule of algorithm $\mathcal{A}$ enabled. This 'scent' is displayed by means of rule T_s and then spread according to rule T_{s2}, and propagated along decreasing values of state parameter c.

Algorithm 3: UNBOUNDED WEAK FAIRNESS UNDER CENTRAL DAEMON

```
T_1:    if $\exists_{u\in N(v)} c(u) = c(v)$
        then $c(v) := 1 + \max\{c(u) \mid u \in N(v)\}$

T_2:    if $c(v) > \min\{0, c(u)+1 \mid u \in N(v) \wedge$
          $c(u) < c(v)\}$
        then $c(v) := \min\{0, c(u)+1 \mid u \in N(v) \wedge$
             $c(u) < c(v)\}$

T_s:    if $\exists_{1\le i\le k} P_{R_i}(v) = true$
        then $s(v) := true$

T_s2:   if $\exists_{u\in N(v)} \big(c(u) > c(v) \wedge s(u) = true\big)$
        then $s(v) := true$

T_3:    if $c(v) = 0 \wedge s(v) = false$
        then $c(v) := 1 + \max\{0, c(u) \mid u \in N(v)\}$;
             $A := true$;
             $s(v) = false$
```

Theorem 5 *If algorithm $\mathcal{A}$ stabilizes under central daemon in $f(n)$ moves then algorithm $\mathcal{T}_A \circ \mathcal{A}$ stabilizes in $O(n^4 \operatorname{diam}(G))f(n)$ steps.*

5 Ensuring weak fairness under distributed daemon

In the previous section, a method for ensuring weak fairness while operating under central daemon was shown. The same task under a distributed daemon seems to be more challenging. Surprisingly, we show that the presented transformations are capable of performing the same task in an environment with accordingly weaker assumptions.

Theorem 6 *If algorithm $\mathcal{A}$ stabilizes under a distributed daemon in $f(n)$ moves then algorithm $\mathcal{T}_A \circ \mathcal{A}$, transformed by Algorithm 2, reaches a safe configuration in $O(n^3 \operatorname{diam}(G))f(n)$ steps.*

Theorem 7 *If algorithm $\mathcal{A}$ stabilizes under a distributed daemon in $f(n)$ moves then algorithm $\mathcal{T}_A \circ \mathcal{A}$, transformed by Algorithm 3, stabilizes in $O(n^4 \operatorname{diam}(G))f(n)$ steps.*

References

[1] J. Beauquier, A. Kumar Datta, M. Gradinariu, F. Magniette, *Self-Stabilizing Local Mutual Exclusion and Daemon Refinement*, Chicago Journal of Theoretical Computer Science, 2002.

[2] J. Beauquier, M. Gradinariu, C. Johnen: *Cross-over composition - enforcement of fairness under unfair adversary*, LNCS 2194, 19–34, 2001.

[3] E.W. Dijkstra, *Self-stabilizing systems in spite of distributed control*, Communications of the ACM **17**, 643–644, 1974.

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

[5] W. Goddard, S.T. Hedetniemi, D.P. Jacobs, P.K. Srimani, *Fault Tolerant Algorithms for Orderings and Colorings*, Proc. IPDPS'04, 2004.

[6] S.T. Hedetniemi, D.P. Jacobs and P.K. Srimani, *Linear time self-stabilizing colorings*, Inform. Process. Lett. **97**, 251-255, 2003.

[7] S.C. Hsu, S.T. Huang, *A self-stabilizing algorithm for maximal matching*, Inform. Process. Lett. **43**, 77–81, 1992.

[8] A. Kosowski, Ł. Kuszner, *A self-stabilizing algorithm for finding a spanning tree in a polynomial number of moves*, LNCS 3911 (to appear).