Probabilistic Snap-Stabilizing Algorithms for Local Resource Allocation Problems

Anaïs Durand, Karine Altisen, Stéphane Devismes

Verimag Research Report n^o TR-2013-6

August 27, 2013

Probabilistic Snap-Stabilizing Algorithms for Local Resource Allocation Problems

Anaïs Durand, Karine Altisen, Stéphane Devismes

August 27, 2013

Abstract

A distributed algorithm is snap-stabilizing if it enables a distributed system to resume a correct behavior immediately after transient faults place it in some arbitrary state. In this paper, we propose two probabilistic snap-stabilizing algorithms to solve the local mutual exclusion problem and their generalizations to some other local resource allocation problems. These algorithms works in any anonymous network. They assume a distributed unfair daemon.

Keywords: Distributed Systems, Probabilistic Snap-Stabilization, Local Resource Allocation

How to cite this report:

```
@techreport {TR-2013-6,
  title = {Probabilistic Snap-Stabilizing Algorithms for Local Resource Allocation Problems},
   author = {Anaïs Durand, Karine Altisen, Stéphane Devismes},
   institution = {{Verimag} Research Report},
   number = {TR-2013-6},
   year = {2013}
}
```

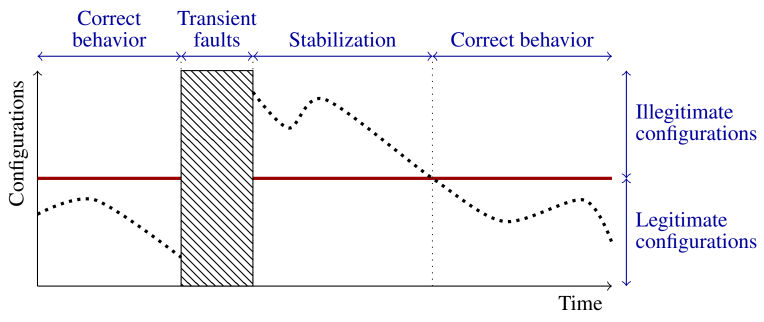


Figure 1: Self-stabilizing system

1 Introduction

A *distributed system* is a set of autonomous entities able to communicate, *e.g.* concurrent processes executed on the same computer, a wired network, or a wireless sensor network. The characteristics of these systems are the absence of knowledge on the global state of the system, the absence of global time, the absence of central control, and the non-determinism of computations due to the asynchronism. Numerous algorithms already exist [8] to solve several kind of problems such as leader election or token-passing.

We consider here *resource allocation* problems [7], where a small number of entities called *resources* should be shared between several processes. The aim is to assign the resources to the processes and managing the access *w.r.t.* compatibility constraints. More precisely, we will consider their restrictions to the neighborhood of a process called *local resource allocation* where the safety is not defined on the global state of the system but on the state of the process and its neighbors.

Due to the scale of distributed systems, it is difficult to envision a human maintenance in case of faults. This is why fault-tolerance is an important requirement. One solution to handle transient faults (*i.e.* faults lasting a finite time and which are infrequent) is *self-stabilization* [10, 11]. An algorithm is self-stabilizing if it can return in a finite time to a *legitimate configuration* (A legitimate configuration is a configuration where the system has a correct behavior. The other configurations are said *illegitimate*) without external help after transient faults lead it to an arbitrary state (FIG. 1).

Many variants of self-stabilization have been proposed. For example, in *probabilistic self-stabilization* [3, 12], the liveness property is weakened to "return in a legitimate state with probability 1" to mainly circumvent impossibility results. Another variant strengthens the safety by ensuring that the system will immediately resume a correct behavior after the end of faults (FIG. 2). This latter approach is called *snap-stabilization* [6]. As for self-stabilization, there is a probabilistic variant of snap-stabilization. In *probabilistic snap-stabilization* [1], the liveness property is ensured with probability 1.

As no deterministic snap-stabilizing solution is possible in an anonymous network, we introduce two new probabilistic snap-stabilizing algorithms to solve some local resource allocation problems in the locally shared memory model under the more general scheduling assumption *i.e.* the distributed unfair daemon. These algorithms can solve for example the local group mutual exclusion problem where two neighbors can concurrently access to the same resource but cannot simultaneously access to two different resources.

2 Model

2.1 Distributed Systems

We consider an undirected (simple) graph $G = (V, E)$ to model the topology of the distributed system. V is a set of n processes, $n \geq 2$. These processes are *anonymous*. In other words, they execute the same program and are not distinguishable if they have the same degree (*e.g.*, they have no unique identifiers). E is a set of edges representing direct communication between processes. If two processes are linked, they can communicate directly and are said *neighbors*. The communications are assumed to be *bidirectional*. $\mathcal{N}_p$ is the set of the neighbors of a process p.

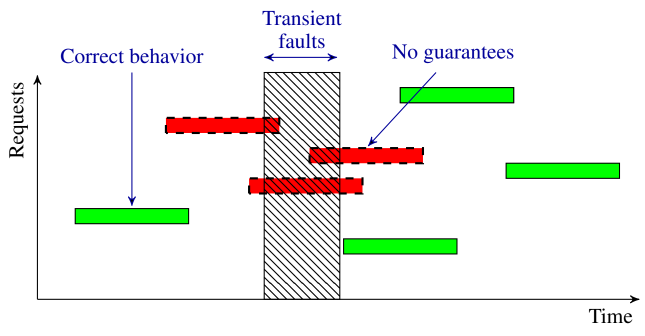


Figure 2: Snap-stabilizing system

2.2 Computational Model

Shared variables: Each process p communicates with its neighbors using a finite number of locally shared registers, called *variables*. p can read and write its own variables, but it can only read the variables of its neighbors. This is the *locally shared memory model*.

The *state* of a process is the vector of the values of all its variables. A *configuration* γ is composed of a state $\gamma(p)$ for each process p. $\gamma(p).v$ is the value of the variable v of p in the configuration γ.

Algorithm: An *algorithm* $\mathcal{A}$ consists of one *program* per process. The program of a process p is a finite set of *actions* of the form:

$$\langle label \rangle : \langle guard \rangle \to \langle statement \rangle$$

The $statement$ updates the variables of p. It can be executed only if the $guard$, a Boolean expression on the variables of p and its neighbors, is evaluated to true in the current configuration γ. Then, this action is said to be enabled in γ. If one or more actions of p are enabled in γ, we say that p is *enabled* in γ. We note $Enabled(\gamma)$ the set of all the processes which are enabled in γ.

Daemon: The *daemon* models the asynchronism of the system.

In a configuration γ, if $Enabled(\gamma)$ is empty, the configuration is said *terminal* and the computation stops. Otherwise, the daemon selects a non-empty subset ϕ of $Enabled(\gamma)$. Each process in ϕ atomically executes one of its enabled actions. Then, the system reaches a new configuration γ'. The transition from γ to γ' is called a *step*.

If $\rho = (\gamma_0, ..., \gamma_k)$ is the finite sequence of configurations that led to the current configuration of the system, $d(\rho) \subseteq Enabled(\gamma_k)$ is the next subset chosen by the daemon.

The sequence $r = (\gamma_i)_{i\geq 0}$, where γ_i are configurations, is a *run* of $\mathcal{A}$ under a daemon d if and only if there exists a sequence of choices of d, $(\phi_i)_{i\geq 0}$, such that $\forall i \geq 0$, the probability to move from the configuration γ_i to the next configuration γ_{i+1} is positive, provided that d has selected the processes in $\phi_i = d(\gamma_0, ..., \gamma_i)$.

Different kinds of daemon exist. They are classified into *families* according to their fairness. Here we consider the family of weakest daemons: the *distributed unfair daemon* family noted $\mathcal{D}_u$. It assumes that: For every run $(\gamma_i)_{i\geq 0}$ of an algorithm $\mathcal{A}$, $\forall i \geq 0$, $Enabled(\gamma_i) \neq \emptyset \Rightarrow d(\gamma_0, ..., \gamma_i) \neq \emptyset$. In other words, if the current configuration is not terminal (at least one process is enabled), the daemon will select at least one process.

2.3 Rounds

We consider a run $r = (\gamma_i)_{i\geq 0}$ of an algorithm $\mathcal{A}$ under a daemon d. If a process p is enabled in a configuration γ_i but not enabled in γ_{i+1} without executing an action between γ_i and γ_{i+1}, we say that p is *neutralized* in the step γ_i, γ_{i+1}.

To evaluate the time complexity of $\mathcal{A}$, we use the notion of *round*.

The minimal prefix of r in which every processes that are enabled in γ_0 either execute an action or become neutralized is the first round of r, noted r'. Let r'' be the suffix of r starting from the last configuration of r'. Then, the first round of r'' is the second round of r and so on.

2.4 Compatibility [7]

In resource allocations problems, several processes share a smaller number of entities called *resource*. A process requests one or several resources. When it gets access to them, it atomically executes a special code called *critical section*. Then, it releases the resources. Let $\mathcal{R}_p$ be the set of resources that can be accessed by a process p.

Two resources X and Y are said *compatible* if two neighbors can concurrently access to them. We note $X \rightleftharpoons Y$. Otherwise, we note $X \not\rightleftharpoons Y$ and X and Y are said *conflicting*. For every resource X, $X \rightleftharpoons \perp$ where $\perp$ symbolizes the absence of request. Note that $\rightleftharpoons$ is symmetrical.

It is possible to define a local resource allocation problem with the relation of compatibility.

Example 1 (Local Mutual Exclusion). In the *local mutual exclusion problem*, two neighbors cannot concurrently access to the unique resource and a process which requests the resource eventually gets it. So, there is only one resource X and $X \not\rightleftharpoons X$.

Example 2 (Local Group Mutual Exclusion). In the *local group mutual exclusion problem*, there are several resources $r_0, r_1, r_2, \ldots, r_k$ shared between the processes. Two neighbors can access concurrently to the same resource but cannot access to different resources at the same time. Then, $\forall i \in \{0, 1, ..., k\}$, $r_i \not\rightleftharpoons r_j$ if and only if $j \neq i$.

Example 3 (Local Readers-Writers). In the *local readers-writers problem*, the processes can access a file in two different ways: a read access (the process is said to be a *reader*) or a write access (the process is a *writer*). A writer must access the file in mutual exclusion, but several readers can concurrently access the file. We represent these two ways of access by two different resources: R for a "read access" and W for a "write access". Then, $R \rightleftharpoons R$ but $R \not\rightleftharpoons W$ and $W \not\rightleftharpoons W$.

3 Specification

First, we recall the definition of probabilistic snap-stabilization.

Definition 1 (Probabilistic Snap-Stabilization [1])**.** An algorithm $\mathcal{A}$ is probabilistically snap-stabilizing *w.r.t* a specification $SP = Safe \cap Live$ and a family of daemons $\mathcal{D}$ if and only if the two following properties are satisfied:

- *Strong Safety:* For each run r under a daemon $d \in \mathcal{D}$, $r \in Safe$.
- *Almost Surely Liveness:* For each run r under a daemon $d \in \mathcal{D}$, $r \in Live$ with probability 1.

3.1 Local Mutual Exclusion Specification

A solution to the local mutual exclusion problem must satisfy the following two properties:

- *Safety:* Two neighbors cannot use the resource at the same time.
- *Liveness:* A process which needs the resource eventually gets it.

3.2 Local Resource Allocation Specification

A solution to some local resource allocation problem expressible with the relation of compatibility must satisfy the following two properties:

- *Safety:* Two neighbors cannot use two conflicting resources at the same time.
- *Liveness:* A process which needs a resource eventually gets it.

3.3 Guaranteed Service Specification

We transform these problems into guaranteed service problems, *i.e.* a problem where an application requests some process to execute finite tasks. Indeed, in a resource allocation problem, when an application at a given process p requests some resources, the service consists in ensuring that the application eventually access to the resources according to the concurrence allowed in the specification of the problem.

In a guaranteed service problem, three properties must be ensured:

1. If an application requests continuously a process to execute the service (here, requests some resources), then the process eventually starts a computation.
2. Every started service eventually ends by a decision at its initiator, allowing it to get back a result. Here, it is the execution of the critical section.
3. Every result obtained from any computation of the service is correct *w.r.t.* the service. In this case, if the critical section is executed, this execution satisfies the safety property of the problem.

To formalize these properties, we use the following predicates, where $p \in V$, s and s' are possible states of p and $\gamma_0, ..., \gamma_t$ are some configurations of the system:

- $Request(s)$ means that the state s indicates to p that some application requests an execution of the service.
- $Start(s, s')$ means that p starts a computation of the service by switching its state from s to s'.
- $Result(s, s')$ means that p executes the decision event to get back the result of a computation by switching its state from s to s'.
- $CorrectResult(\gamma_0 \dots \gamma_t, p)$ means that the computed result is correct *w.r.t* the service, *i.e.* the execution of the critical section is in local mutual exclusion.

Then we have the following specification: $S = Safe \cap Live$ where $Safe$ and $Live$ are defined as follows for $r = (\gamma_i)_{i \geq 0}$:

- $r \in Safe$ if and only if $\forall k \geq 0, \forall p \in V, (Result(\gamma_k(p), \gamma_{k+1}(p)) \wedge \exists l < k, Start(\gamma_l(p), \gamma_{l+1}(p))) \Rightarrow CorrectResult(\gamma_0 \dots \gamma_k, p)$.

- $r \in Live$ if and only if the two following conditions are ensured:
  - $\forall k \geq 0, \forall p \in V, \exists l > k, (Request(\gamma_l(p)) \Rightarrow Start(\gamma_l(p), \gamma_{l+1}(p)))$.
  - $\forall k \geq 0, \forall p \in V, (Start(\gamma_k(p), \gamma_{k+1}(p)) \Rightarrow (\exists l > k, Result(\gamma_l(p), \gamma_{l+1}(p))))$.

3.4 Guaranteed Service Local Mutual Exclusion Specification

To specify the guaranteed service local mutual exclusion problem, $CorrectResult$ is instantiated as follows:

$$CorrectResult(\gamma_0 \dots \gamma_k, p) \equiv Computing(\gamma_k(p)) \Rightarrow (\forall q \in \mathcal{N}_p, \neg Computing(\gamma_k(q)))$$

$Computing(s)$ is a predicates. It means that p is allowed to execute its critical section.

3.5 Guaranteed Service Local Resource Allocation Specification

To specify the guaranteed service local resource allocation problem, $CorrectResult$ is instantiated as follows:

$$\begin{aligned} CorrectResult(\gamma_0 \dots \gamma_k, p) \equiv & Computing(\gamma_k(p)) \Rightarrow (\forall q \in \mathcal{N}_p, \neg Computing(\gamma_k(q)) \\ & \vee \gamma_k(p).request \rightleftharpoons \gamma_k(q).request) \end{aligned}$$

$Computing(s)$ is a predicates. It means that p is allowed to execute its critical section.

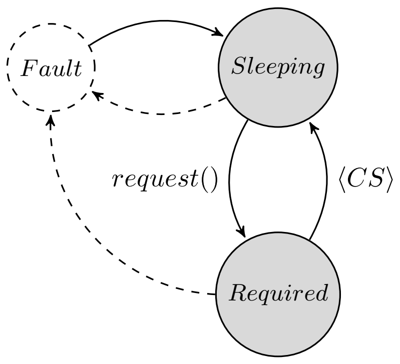


Figure 3: Automaton of the interface between the application and the algorithm

3.6 Interface between the application and the algorithm

The interface between the application and the algorithm can be represented by the automaton on FIG. 3.

When a process is $Sleeping$, if the application requests it and if it switches its state from s to s', then $Request(s) \wedge Start(s, s')$ is true and it goes to $Required$. When the critical section $\langle CS \rangle$ is executed by switching from the state t to t', $Result(t, t')$ is true and the process comes back to $Sleeping$. Some faults can occur and put it in $Fault$, but the process will come back to $Sleeping$ in a finite number of steps.

If the application requests continuously the process, even if it is currently in $Fault$, it eventually goes to $Sleeping$, then to $Required$ and executes the critical section.

4 First Probabilistic Snap-Stabilizing Algorithm For Local Mutual Exclusion Problem

This algorithm is composed of two parts: the core of the algorithm (ALGO. 2) and a synchronization scheme called *unison* (ALGO. 1).

4.1 Unison

The *unison* proposed in [5] is a self-stabilizing algorithm that synchronizes the processes with *logical clocks*. As it is impossible to exactly synchronize processes under a distributed unfair daemon, the synchronization property is weakened to synchronize with a difference of at most one between the clocks of two neighbors. Let make a review of the scheme in [5].

Each process has an integer variable called clock. In a legitimate configuration, *i.e.* when $AllCorrect$ is true for each process, and when the process has the smaller clock among its neighbors, it increments its clock by using the action NA. This ensures that a process can only make a finite number of incrementations of its clock before an incrementation of the clocks of its neighbors. In an illegitimate state, the goal is to propagate a reset to re-synchronize the system and resume a correct behavior. When a process detects the error, it resets its clock to $-\alpha$ by the action RA. Then the process re-synchronizes with its neighbors during the *reset phase* using the action CA.

4.2 Synchronization Between the Two Parts of the Algorithm

The local mutual exclusion algorithm (ALGO. 2) uses this scheme and executes the actions of the synchronization scheme plus its own actions during a NA action when it is possible. More precisely, ALGO. 2 can be re-written using ALGO. 1 the following way:

- CA and RA actions remain identical.
- For all action $A_i : guard_i \Rightarrow stmt_i$ of ALGO. 2, it is replaced by:

$$A_i : NormalStep_p \wedge guard_i \Rightarrow c_p := \varphi(c_p);\ stmt_i;$$

Algorithm 1 General synchronization scheme [5]

```
Constants
    $N$: an upper bound on the size of the network
Variables
    $c_p \in \mathcal{X} = \{-\alpha, \dots, 0, \dots, K-1\}, \quad \alpha \geq N-2, \quad K \geq N$
Functions
    $\varphi : x \to \begin{cases} (x+1) & \text{if } x < 0 \\ (x+1) \pmod K & \text{otherwise} \end{cases}$
  Predicates
    $Correct_p(q)$           $\equiv$  $(\varphi(c_p) = c_q) \vee (c_p = c_q) \vee (c_p = \varphi(c_q))$
    $AllCorrect_p$           $\equiv$  $\forall q \in \mathcal{N}_p, Correct_p(q)$
    $NormalStep_p$           $\equiv$  $(c_p \geq 0) \wedge (\forall q \in \mathcal{N}_p, c_q \geq 0 \wedge (\varphi(c_p) = c_q \vee c_p = c_q))$
    $ConvergenceStep_p$      $\equiv$  $(c_p < 0) \wedge (\forall q \in \mathcal{N}_p, (c_q \leq 0) \wedge (c_p \leq c_q))$
    $ResetInit_p$            $\equiv$  $(c_p > 0) \wedge \neg UnisonCorrect_p$
Actions
    $NA$ :  $NormalStep_p$       $\to$  $c_p := \varphi(c_p)$;
    $CA$ :  $ConvergenceStep_p$  $\to$  $c_p := \varphi(c_p)$;
    $RA$ :  $ResetInit_p$        $\to$  $c_p := -\alpha$;
```

- NA action is replaced by:

$$NA : NormalStep_p \wedge \neg guard_0 \wedge \neg guard_1 \wedge \cdots \wedge \neg guard_k \Rightarrow c_p := \varphi(c_p);$$

4.3 Local Mutual Exclusion Algorithm

States: Each process has four states. A node in the state:

- $IDLE$ is not using the resource and is not requesting it.
- TRY wants the resource, but does not get it already. It may be in competition with its neighbors.
- WIN wants the resource and will get it if it is the only one in this state in the neighborhood.
- IN is using the resource.

Actions in a legitimate configuration: There are two kinds of actions. First, let study the actions when the system is in a legitimate configuration.

When a node needs the resource, it jumps from the state $IDLE$ to the state TRY (FIG. 4) using the action Req. Then with probability $\mathsf{p} \in (0, 1)$ it stays in the state TRY and with probability $1 - \mathsf{p}$, it jumps to the state WIN (FIG. 5) executing the action $Rand$.

If it is the only one in the state WIN and if no one in its neighborhood is in the state IN, it can get the resource (FIG. 6) with the action $Enter$. Otherwise, it tries one more time (FIG. 5) using the action $Rand$.

When a node has gotten the access to the resource, it comes back to the state $IDLE$ and executes its critical section (FIG. 7) executing the action Rel.

Global view: To sum up, if the node wants the resource, it goes to the state TRY and cannot come back to the state $IDLE$ without going first to the state IN and getting the resource. Moreover, when a node is in the state TRY, it must compete with its neighbors which also want the resource and it can get it immediately.

Let us take the case where the node must compete with only one of its neighbors (this can be generalized to any number of nodes in competition). Thanks to the synchronization scheme, if a node is activated, it can be activated at most one time again until all its neighbors increment their clocks. So each node will be activated infinitely often. Hence the node which "tries" to obtain the resource will be activated infinitely often in order to play for winning. Due to the probability law, it will win after a while with probability 1 and get the resource.

Algorithm 2 Local mutual exception algorithm

```
p ∈ $(0,1)$
Inputs
    $\langle critical\ section\rangle$: code of the critical section
    $?p.request()$: function from the application
Outputs
    $!p.get()$
    $!p.release()$
Variables
    $s_p \in \{IDLE, TRY, WIN, IN\}$
Predicates
    $NeighborIn_p$        $\equiv$  $\exists q \in \mathcal{N}_p, s_q = IN$
    $NeighborWin_p$       $\equiv$  $\exists q \in \mathcal{N}_p, s_q = WIN$
    $RequestStep_p$       $\equiv$  $?p.request() \wedge (s_p = IDLE)$
    $ReleaseStep_p$       $\equiv$  $(s_p = IN) \wedge \neg NeighborIn_p$
    $RandomStep_p$        $\equiv$  $\neg NeighborIn_p \wedge ((s_p = TRY \wedge \neg NeighborWin_p)$
                                      $\vee (s_p = WIN \wedge NeighborWin_p))$
    $EnterStep_p$         $\equiv$  $(s_p = WIN) \wedge \neg NeighborIn_p \wedge \neg NeighborWin_p$
    $ResetWinStep_p$      $\equiv$  $(s_p = WIN) \wedge NeighborIn_p$
    $ResetStateStep_p$    $\equiv$  $(s_p = IN) \wedge NeighborIn_p$
Actions
    $Req$ :     $RequestStep_p$     $\rightarrow$  $s_p := TRY;$
    $Rel$ :     $ReleaseStep_p$     $\rightarrow$  $!p.get();\ \langle critical\ section\rangle;\ !p.release();\ s_p := IDLE;$
    $Rand$ :    $RandomStep_p$      $\rightarrow$  $s_p := \begin{cases} TRY; & \text{with probability p} \\ WIN; & \text{with probability } 1-\text{p} \end{cases}$
    $Enter$ :   $EnterStep_p$       $\rightarrow$  $s_p := IN;$
    $RstW$ :    $ResetWinStep_p$    $\rightarrow$  $s_p := TRY;$
    $RstI$ :    $ResetStateStep_p$  $\rightarrow$  $s_p := IDLE;$
```

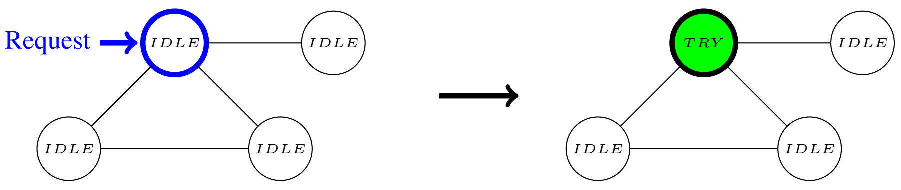


Figure 4: Req rule

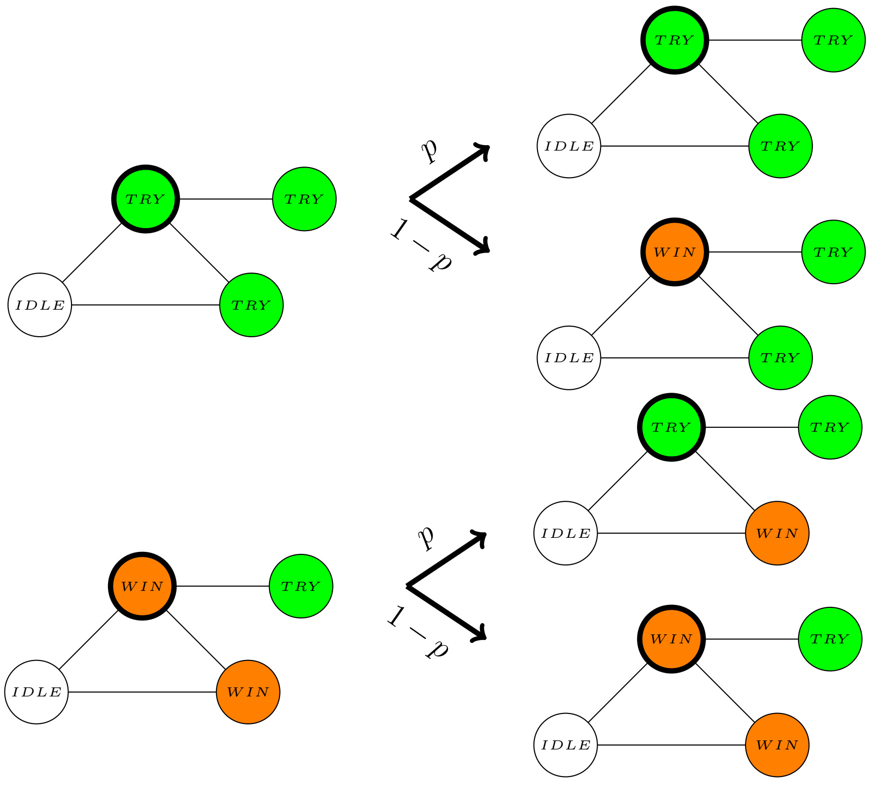


Figure 5: Rand rule

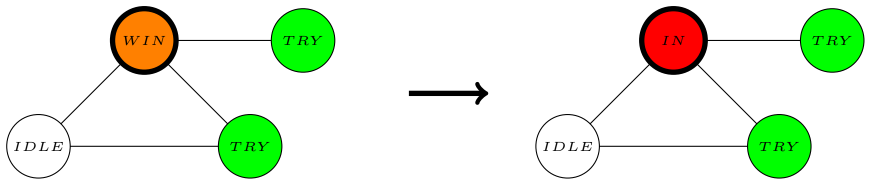


Figure 6: Enter rule

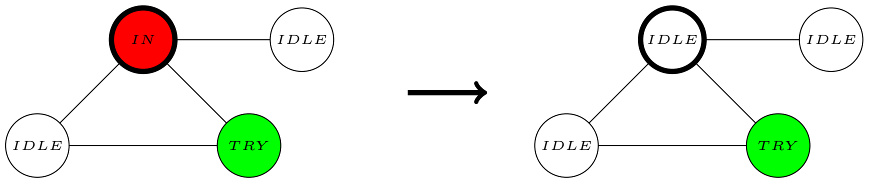


Figure 7: Rel rule

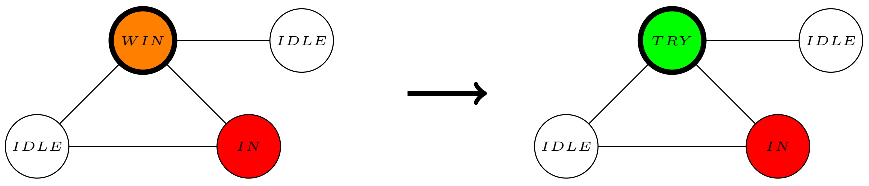


Figure 8: RstW rule

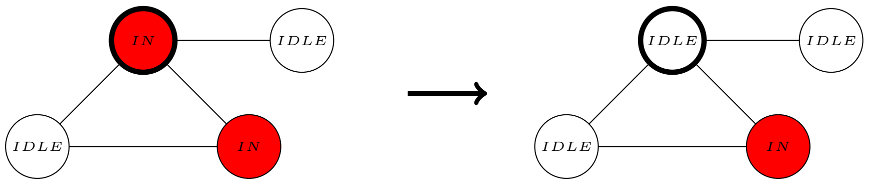


Figure 9: RstI rule

Actions in an illegitimate configuration: Transient faults can occur and affect the memory of the processes. Two actions are then used to recover :

- If a node is in the state WIN while one of its neighbors is using the resource, it comes back to the state TRY (FIG. 8) using the action $RstW$.
- If a node is in the state IN while one of its neighbors is also using the resource, it comes back to the state $IDLE$ (FIG. 9) using the action $RstI$.

4.4 Example (FIG. 10)

Let see a small example of the execution of the algorithm in a synchronous context for more simplicity.

Two processes, which are neighbors, are requested to get the resource (FIG. (a)) and will compete to get it (FIGS. (b), (c), (d)). One of them will get it first (FIG. (e)). When it executes its critical section and releases the resource (FIG. (f)), the second will be able to use it (FIG. (g)).

In FIG. (f), we call x the process which released the resource and y the process which wants the resource. If x is requested again, the unfair daemon has no constraints and can select only x until it wins against y and gets the resource again. This situation could occur and could be repeated endlessly if ALGO. 1 is not take into account. But, the unison ensures that x can only “play” a finite number of time until a try of y. So, y will have a chance to get the resource.

4.5 Instantiation of the specification by the algorithm:

ALGO. 2 instantiates the guaranteed service predicates as follows:

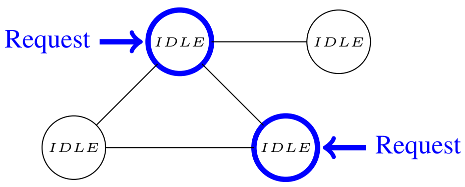
- $Request(s) \equiv s.request()$, where $request()$ is a function of the application.
- $Start(s, s') \equiv s.s = IDLE \wedge s'.s = TRY$, *i.e.* p executes Req by switching its state from s to s'.
- $Result(s, s') \equiv s.s = IN \wedge \neg s.NeighborIn \wedge s'.s = IDLE$, *i.e.* p executes Rel by switching its state from s to s'.
- $Computing(s) \equiv s.s = IN$

4.6 Proof

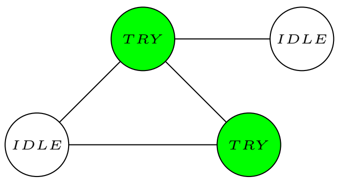
Assume an execution of the system under a daemon d, where the requests are set and with any initial configuration γ_0.

Definition 2 (Synchronized)**.** The clocks in a configuration γ are *synchronized* and we note $Synchronized(\gamma)$ if and only if $\forall p \in V, \forall q \in \mathcal{N}_p, UnisonCorrect(p, q, \gamma)$.

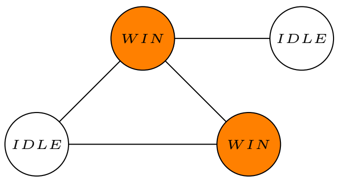
$UnisonCorrect(p, q, \gamma)$ is true if $(\gamma(p).c = \varphi(\gamma(q).c)) \vee (\gamma(p).c = \gamma(q).c) \vee (\varphi(\gamma(p).c) = \gamma(q).c)$.



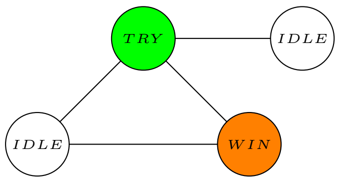
(a) Two neighbors are requested to get the resource.



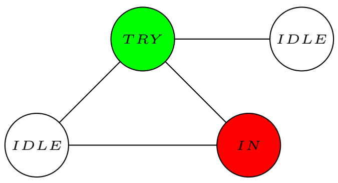
(b) They must compete and “flip a coin” to choose if they stay in the state TRY or jump to the state WIN.



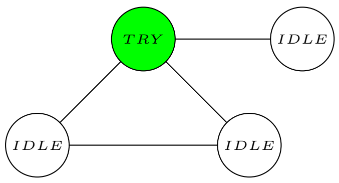
(c) They have to try again.



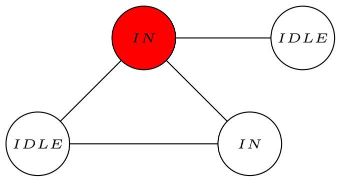
(d) One of them wins because it is the only one in its neighborhood being in the state WIN.



(e) It can take the resource. The other must wait.



(f) The process which has get the access to the resource executes its critical section and releases it.



(g) The other process can take the resource.

Figure 10: Small example

Definition 3 (Locally correct)**.** A process p in a configuration γ is *locally correct* if and only if $\forall q \in \mathcal{N}_p, (\gamma(p).s = IN \vee \gamma(p).s = WIN) \Rightarrow \gamma(q).s \neq IN$. We note $LocallyCorrect(p, \gamma)$.

Lemma 1. *In a step from γ to γ', if a process p is locally correct in γ then p is locally correct in γ'.*

Proof. Let $p \in V$ and γ a configuration where p is locally correct. We prove that p is still locally correct in the configuration γ' after the execution of a step.

Assume that p is not locally correct in γ'. Then, there are two cases:

1. $\gamma'(p).s = IN \wedge \exists q \in \mathcal{N}_p, \gamma'(q).s = IN$.

   There is no action to jump from the state $IDLE$ or the state TRY to the state IN (in one step). Hence, $\gamma(p).s$ and $\gamma(q).s$ are different from $IDLE$ and TRY.

   Let study the other cases:

   (a) $\gamma(p).s = IN$: If $\gamma(q).s = IN$ then p is not locally correct in γ. It is a contradiction. If $\gamma(q).s = WIN$ then the only action that q can make to jump to the state IN is $Enter$, but this action is enabled only if $\nexists n \in \mathcal{N}_q, \gamma(n).s = IN$. It is a contradiction.

   (b) $\gamma(p).s = WIN$: If $\gamma(q).s = IN$ then p is not locally correct in γ. It is a contradiction. If $\gamma(q).s = WIN$, q can only make the action $Enter$ to jump into the state IN, but this action is enabled only if $\nexists n \in \mathcal{N}_q, \gamma(n).s = WIN$. It is a contradiction.

2. $\gamma'(p).s = WIN \wedge \exists q \in \mathcal{N}_p, \gamma'(q).s = IN$.

   There is no action to jump in one step from the states $IDLE$ or TRY to IN and there is no action to jump from the states $IDLE$ or IN to WIN in one step. Hence, $\gamma(p).s \neq IDLE \wedge \gamma(p).s \neq IN$ and $\gamma(q).s \neq IDLE \wedge \gamma(q).s \neq TRY$.

   Let study the other cases:

   (a) $\gamma(p).s = TRY$: The only action that p can make to go to the state WIN is $Rand$ but this action is enabled only if $\nexists n \in \mathcal{N}_p, \gamma(n).s = WIN \vee \gamma(n).s = IN$. This is a contradiction.

   (b) $\gamma(p).s = WIN$: See the case 1b.

Hence, p is locally correct in γ'. □

Lemma 2. *If p executes an action from γ to γ' and p is not locally correct in γ then p is locally correct in γ'.*

Proof. If p is not locally correct in γ, there are two cases:

1. $\gamma(p).s = IN \wedge (\exists q_1 \in \mathcal{N}_p, \gamma(q_1).s = IN)$: The only action that can execute p is $RstI$. So $\gamma'(p).s = IDLE$ and p is locally correct in γ'.

2. $\gamma(p).s = WIN \wedge (\exists q_2 \in \mathcal{N}_p, \gamma(q_2) = IN)$: The only action that can execute p is $RstW$. So $\gamma'(p) = TRY$ and p is locally correct in γ'.

□

Definition 4 (Legitimate configuration)**.** A configuration γ is *legitimate* if and only if $Synchronized(\gamma) \wedge (\forall p \in V, LocallyCorrect(p, \gamma))$.

Lemma 3. *The set of legitimate configurations is closed.*

Proof. Let γ be a legitimate configuration and γ' a configuration reachable from γ in one step. We prove that γ' is legitimate.

Assume that γ' is illegitimate. The synchronization scheme is self-stabilizing [5] so $\forall p \in V, \forall q \in \mathcal{N}_p, UnisonCorrect(p, q, \gamma')$. Then, $\exists p \in V, \neg LocallyCorrect(p, \gamma')$. As γ is legitimate, p is locally correct in γ. But, by Lemma 1, $LocallyCorrect(p, \gamma) \Rightarrow LocallyCorrect(p, \gamma')$. It is a contradiction.

Hence, γ' is legitimate. □

Corollary 1. *A run $(\gamma_i)_{i\geq 0}$ where γ_0 is legitimate two neighbors cannot be in IN simultaneously.*

Lemma 4. *A system in an illegitimate configuration converges to a legitimate configuration in at most $O(n)$ rounds.*

Proof. The unison satisfies the following properties: If the system is in an illegitimate configuration, the clocks synchronize in at most $O(n)$ rounds [4]. Moreover, in at most $\mathcal{D}+1$ rounds a process increments its clock [2] (where $\mathcal{D}$ is the diameter of the network).

So, after the synchronization of the clocks in at most $O(n)$ rounds, each not locally correct process p increments its clock and executes an action that leads it to become locally correct (Lemma 2) in at most $\mathcal{D}+1$ rounds. Then p stays locally correct (Lemma 1). Therefore, in at most $\mathcal{D}+1$ rounds, all processes are locally correct.

To conclude, in at most $O(n)$, the system is in a legitimate configuration. □

Lemma 5. *A system in an illegitimate configuration converges to a legitimate configuration in at most $O(\mathcal{D}n^3)$ steps, where $\mathcal{D}$ is the diameter of the network.*

Proof. The synchronization scheme satisfies the following properties: It converges in at most $O(\mathcal{D}n^3)$ steps [9] and a process will increment its clock after at most $2\mathcal{D}(n-1)+1$ steps [2].

Then the proof is the same than the proof for Lemma 4. □

Definition 5 (Annoying neighbor)**.** In a configuration γ, let p be a process such that $\gamma(p).s = TRY \vee \gamma(p).s = WIN$. A neighbor $q \in \mathcal{N}_p$ of p is said *annoying* for p if $\gamma(q).s = IN \vee \gamma(q).s = WIN$. Otherwise, q is said *non-annoying* for p. We note $Annoying_p(\gamma)$ the set of annoying neighbors of p in γ.

Lemma 6. *In a legitimate configuration, with a positive probability, a non-annoying neighbor will stay non-annoying after a step.*

Proof. In a configuration γ, if $q \in \mathcal{N}_p$ is not an annoying neighbor for p then there are two cases:

- $\gamma(q).s = IDLE$: q executes Req and goes to the state TRY. Otherwise, q stays in $IDLE$.
- $\gamma(q).s = TRY$: If $\exists x \in \mathcal{N}_q, \gamma(x).s = IN \vee \gamma(x).s = WIN$, then q stays in the state TRY and only increments its clock, else executing $Rand$, q stays in the state TRY with probability $\mathsf{p} > 0$.

□

Lemma 7. *In a legitimate configuration, with a positive probability, an annoying neighbor will become non-annoying after at most two steps.*

Proof. In a configuration γ, if $q \in \mathcal{N}_p$ is an annoying neighbor for p then there are two cases:

- $\gamma(q).s = IN$: q executes Rel and jumps to the state $IDLE$. Then, with a positive probability, it stays non-annoying (Lemma 6).
- $\gamma(q).s = WIN$: If $\nexists x \in \mathcal{N}_q, \gamma(x).s = WIN$, q executes $Enter$, goes to the state IN and then goes to the state $IDLE$ executing Rel. Else, q goes to the state TRY with probability $\mathsf{p} > 0$ executing $Rand$ and stays in this state with probability $\mathsf{p} > 0$ executing $Rand$ another time.

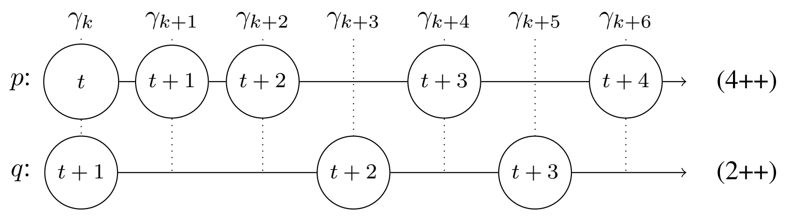
□

Lemma 8. *In a configuration γ such that $Synchronized(\gamma)$ is true, after four incrementations of the clock of p, its neighbors increment at least 2 times their clocks.*

Proof. Let $p \in V$ and a run $r = (\gamma_i)_{i\geq 0}$ under a daemon d. Assume that γ_k is a configuration where p makes an incrementation of its clock and let $\gamma_k(p).c = t$. We try to increment the clock of q as seldom as possible.

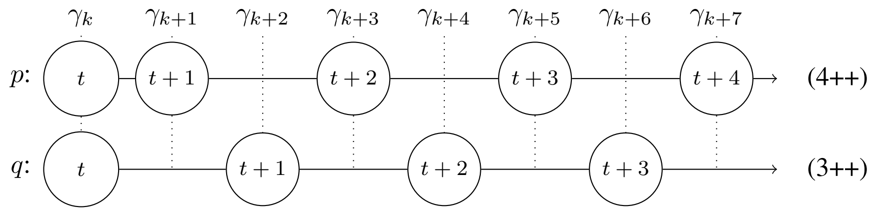
Let $q \in \mathcal{N}_p$. Examine the different cases:

- If $\varphi(\gamma_k(p).c) = \gamma_k(q).c$:



  In this case, q must increment its clock at least 2 times.

- If $\gamma_k(p).c = \gamma_k(q).c$:



  In this case, q must increment its clock at least 3 times.

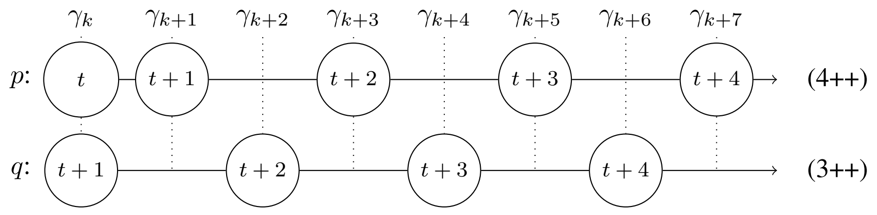
□

Lemma 9. *In a configuration γ such that $Synchronized(\gamma)$ is true, after four incrementations of the clock of p, its neighbors increment at most 4 times their clocks.*

Proof. Let $p \in V$ and a run $r = (\gamma_i)_{i\geq 0}$ under a daemon d. Assume that γ_k is a configuration where p makes an incrementation of its clock and let $\gamma_k(p).c = t$. We try to increment the clock of q as often as possible.

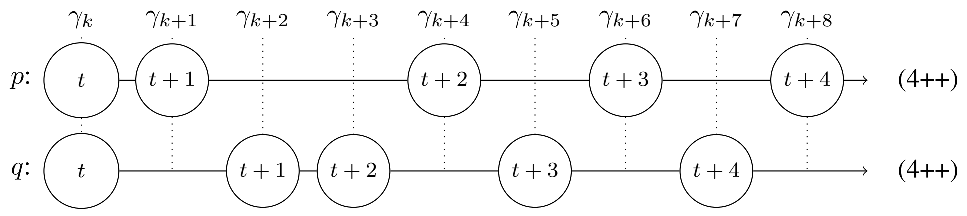
Let $q \in \mathcal{N}_p$. Examine the different cases:

- If $\varphi(\gamma_k(p).c) = \gamma_k(q).c$:



  In this case, q can increment its clock at most 3 times.

- If $\gamma_k(p).c = \gamma_k(q).c$:

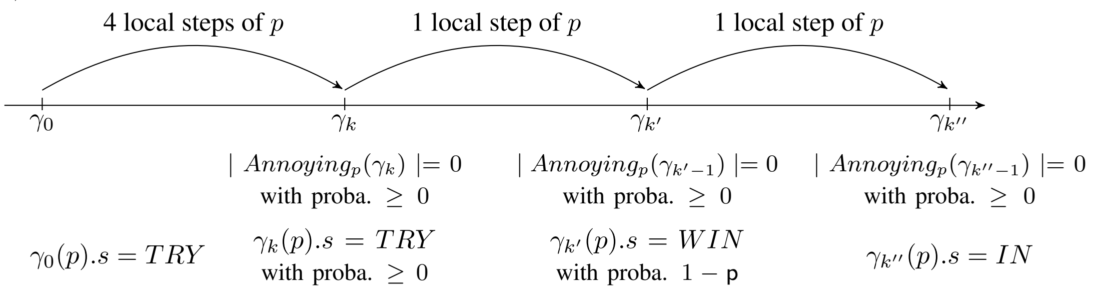


  In this case, q can increment its clock at most 4 times.

□

Lemma 10. *If a process is requested to use the resource in a legitimate configuration, it will almost surely execute its critical section in a finite time.*

Proof. Let $p \in V$. We exhibit a run $r = (\gamma_i)_{i \geq 0}$ which occurs with positive probability and for which p is served after 6 local steps (*i.e.* the execution of 6 actions by p). Assume that γ_0 is legitimate and that $\gamma_0(p).s = TRY$.


4 local steps of p
1 local step of p
1 local step of p
γ_0
γ_k
$\gamma_{k'}$
$\gamma_{k''}$
$\mid Annoying_p(\gamma_k) \mid = 0$ with proba. ≥ 0
$\mid Annoying_p(\gamma_{k'-1}) \mid = 0$ with proba. ≥ 0
$\mid Annoying_p(\gamma_{k''-1}) \mid = 0$ with proba. ≥ 0
$\gamma_0(p).s = TRY$
$\gamma_k(p).s = TRY$ with proba. ≥ 0
$\gamma_{k'}(p).s = WIN$ with proba. $1 - \mathsf{p}$
$\gamma_{k''}(p).s = IN$

We note γ_k the configuration after which p executed four actions. Using Lemmas 8 and 9, all the neighbors of p executed between two and four actions between γ_0 and γ_k. Using Lemma 7, an annoying neighbor becomes non-annoying after the execution of at most two steps with a positive probability and then stays non-annoying with a positive probability (Lemma 6). Moreover, if p has annoying neighbors, it deterministically stays in TRY. Otherwise, it stays in TRY with probability $\mathsf{p} > 0$ by executing $Rand$. So with a positive probability, $\mid Annoying_p(\gamma_k) \mid = 0$ and $\gamma_k(p).s = TRY$.

We note $\gamma_{k'}$ the configuration after γ_k and after which p executed a new action. Using Lemma 6, $\mid Annoying_p(\gamma_{k'-1}) \mid = 0$ with a positive probability. Then p executes between $\gamma_{k'-1}$ and $\gamma_{k'}$ the action $Rand$ and $\gamma_{k'}(p).s = WIN$ with probability $1 - \mathsf{p} > 0$.

We note $\gamma_{k''}$ the configuration after $\gamma_{k'}$ and after which p executed a new action. Using Lemma 6, $\mid Annoying_p(\gamma_{k''-1}) \mid = 0$ with a positive probability. Then p executes between $\gamma_{k''-1}$ and $\gamma_{k''}$ the action $Enter$ and $\gamma_{k''}(p).s = IN$.

So after $s = k''$ steps, there is a positive probability (let call it p') that p is served. s is a constant fixed by the daemon and unison. Then:

$$
\begin{aligned}
P(p \text{ is not served after } s \text{ steps}) &= 1 - P(p \text{ is served after } s \text{ steps}) \\
&\leq 1 - \mathsf{p}' \\
P(p \text{ is not served after } 2s \text{ steps}) &\leq (1 - \mathsf{p}')^2 \\
&\;\vdots \\
P(p \text{ is never served}) &= \lim_{x \to \infty} P(p \text{ is not served after } xs \text{ steps}) \\
&\leq \lim_{x \to \infty} (1 - \mathsf{p}')^x = 0 \qquad \text{since } \mathsf{p}' \text{ is positive} \\
P(p \text{ is served}) &= 1 - P(p \text{ is never served}) = 1
\end{aligned}
$$

□

Theorem 1. ALGO. 2 satisfies the probabilistic snap-stabilizing guaranteed service local mutual exclusion specification.

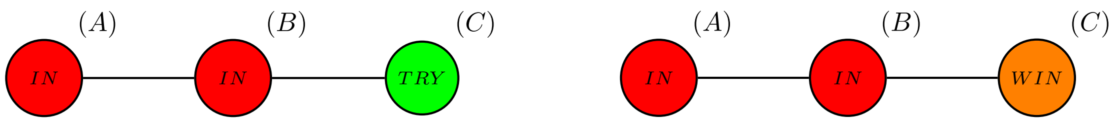
Proof. Let $r = (\gamma_i)_{i \geq 0}$ under a daemon d.

- $r \in Safe$: $\forall k \geq 0, \forall p \in V$, if $Result(\gamma_k(p), \gamma_{k+1}(p))$, then $\gamma_k(p).s = IN \wedge \neg \gamma_k(p).NeighborIn \wedge \gamma_{k+1}(p).s = IDLE$. Then, $Computing(\gamma_k(p)) \wedge \forall q \in \mathcal{N}_p, \neg Computing(\gamma_k(q))$.

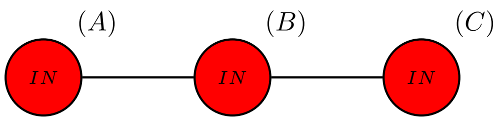
  So $CorrectResult(\gamma_0 \ldots \gamma_k, p)$ is true by definition.

- $r \in Live$ with probability 1:

  - $\forall k \geq 0, \forall p \in V$, if $\gamma_k(p).s \neq IDLE$, p is computing a service and cannot answer to a new request. But, in a finite time, p will almost surely end its current computation and come back to the state $IDLE$ (Lemma 10). Then, after a finite time, p will be selected by the daemon to execute an action in a configuration γ_l (with $l \geq k$). If the application still request the



(a) (C) has only one neighbor in IN so it can execute $Rand$ (b) And then it can enter into IN using $Enter$



(c) But the safety is violated for (B)

Figure 11: Example for the local 2-exclusion problem

resource, the guard of Req is true ($\gamma_l(p).s = IDLE \wedge \gamma_l(p).request()$) and the result is $\gamma_{l+1}(p).s = TRY$. So $Start(\gamma_l(p), \gamma_{l+1}(p))$ is true.

- $\forall k \geq 0, \forall p \in V$, if $Start(\gamma_k(p), \gamma_{k+1}(p))$ then $\gamma_{k+1}(p).s = TRY$. Using Lemma 10, in a finite time, p will almost surely compute its critical section, *i.e.* execute Rel. Then, $\exists l > k$ such as the guard of Rel is true ($\gamma_l(p).s = IN \wedge \neg\gamma_l(p).NeighborIn$) and the result of the execution of Rel is $\gamma_{l+1}(p).s = IDLE$. So $Result(\gamma_l(p), \gamma_{l+1}(p))$ is true.

□

5 Generalization To Some Other Local Resource Allocation Problems

In this section, we introduce a generalization of our algorithm to some other local resource allocation problems.

ALGO. 3 is the generalization of the algorithm of local mutual exclusion to the problems expressible with $\rightleftharpoons$. Each process p as a variable $request_p$ which is an input from the application. In this variable, there is the resource requested by the application. Instead of checking if there are neighbors in IN or WIN, a process checks if the requests of its neighbors in IN or WIN are compatible with its own request.

Note that a process p such that $s_p \neq IDLE$ and $request_p = \perp$ is not considered as an error even if this situation cannot occurs in the normal behavior of the algorithm. If this case occur after a fault, p will have the same behavior as a process which requests a resource without annoying its neighbors because $\forall R, \perp \rightleftharpoons R$.

5.1 Local l-Exclusion and Local k-Out-Of-l-Exclusion

ALGO. 3 does not solve the local l-exclusion and the local k-out-of-l-exclusion problems. These problems need that a process can have a view of the configuration of the system at distance 2.

For example, FIG. 11 is an example of what happens with the local 2-exclusion problem. In FIG. (a), (C) has only one neighbor in IN so it can go to WIN (FIG. (b)) and then to IN (FIG. (c)). But (B) already had a neighbor in IN. With (C) there is 3 nodes in IN in its neighborhood, so the safety is violated.

In order to enter in IN, (C) must know that (B) has no other neighbors in IN or wait for an authorization of (B) to enter.

5.2 Instantiation of the specification by the algorithm:

ALGO. 3 instantiates the guaranteed service predicates as follows:

Algorithm 3 Generalization

```
p ∈ (0, 1)
Inputs
    $\langle critical\ section \rangle$: code of the critical section
    $?request_p \in \mathcal{R}_p$: input from the application
Outputs
    $!p.get()$
    $!p.release()$
Variables
    $s_p \in \{IDLE, TRY, WIN, IN\}$
Predicates
    $CompatibleWithIn_p$          $\equiv$  $\forall q \in \mathcal{N}_p, (s_q = IN) \Rightarrow (request_p \rightleftharpoons request_q)$
    $CompatibleWithInAndWin_p$    $\equiv$  $\forall q \in \mathcal{N}_p, (s_q = IN \vee s_q = WIN$
                                              $\Rightarrow request_p \rightleftharpoons request_q)$
    $RequestStep_p$               $\equiv$  $?request_p \neq \bot \wedge (s_p = IDLE)$
    $ReleaseStep_p$               $\equiv$  $(s_p = IN) \wedge CompatibleWithIn_p$
    $RandomStep_p$                $\equiv$  $(s_p = TRY \wedge CompatibleWithInAndWin_p)$
                                              $\vee(s_p = WIN \wedge \neg CompatibleWithInAndWin_p)$
    $EnterStep_p$                 $\equiv$  $(s_p = WIN) \wedge CompatibleWithInAndWin_p$
    $ResetWinStep_p$              $\equiv$  $(s_p = WIN) \wedge \neg CompatibleWithIn_p$
    $ResetStateStep_p$            $\equiv$  $(s_p = IN) \wedge \neg CompatibleWithIn_p$
Actions
    $Req$ :     $RequestStep_p$            $\rightarrow$  $s_p := TRY;$
    $Rel$ :     $ReleaseStep_p$            $\rightarrow$  $!p.get();\ \langle critical\ section \rangle;\ !p.release();$
                                                $request_p = \bot;\ s_p := IDLE;$
    $Rand$ :    $RandomStep_p$             $\rightarrow$  $s_p := \begin{cases} TRY; & \text{with probability p} \\ WIN; & \text{with probability } 1 - \text{p} \end{cases}$
    $Enter$ :   $EnterFromWinStep_p$       $\rightarrow$  $s_p := IN;$
    $RstW$ :    $ResetWinStep_p$           $\rightarrow$  $s_p := TRY;$
    $RstI$ :    $ResetStateStep_p$         $\rightarrow$  $request_p = \bot;\ s_p := IDLE;$
```

- $Request(s) \equiv s.request() \neq \perp$, where $request()$ is a function of the application.
- $Start(s, s') \equiv s.s = IDLE \wedge s'.s = TRY$, *i.e.* p executes Req by switching its state from s to s'.
- $Result(s, s') \equiv s.s = IN \wedge s.CompatibleWithIn \wedge s'.s = IDLE$, *i.e.* p executes Rel by switching its state from s to s'.
- $Computing(s) \equiv s.s = IN$

5.3 Proof of the Generalization

We go back over the definitions and proofs of lemmas that change due to the generalization. The other definitions, lemmas or proofs stay identical.

Definition 6 (Locally correct)**.** A process p in a configuration γ is *locally correct* if and only if $\forall q \in \mathcal{N}_p, (\gamma(p).s = IN \vee \gamma(p).s = WIN) \Rightarrow \gamma(p).CompatibleWithIn$. We note $LocallyCorrect(p, \gamma)$.

Lemma 11. *In a step from γ to γ', if a process p is locally correct in γ then p is locally correct in γ'.*

Proof. Let $p \in V$ and γ a configuration where p is locally correct. We prove that p is still locally correct in the configuration γ' after the execution of a step.

Assume that p is not locally correct in γ'. Then, there are two cases:

1. $\gamma'(p).s = IN \wedge \neg\gamma'(p).CompatibleWithIn$.

   Then $\exists q \in \mathcal{N}_p, \gamma'(q).s = IN \wedge \gamma'(p).request \not\Rightarrow \gamma'(q).request$. There is no action to jump from the state $IDLE$ or the state TRY to the state IN (in one step). Hence, $\gamma(p).s$ and $\gamma(q).s$ are different from $IDLE$ and TRY.

   Note that as p and q cannot execute their critical section between γ and γ', their requests do not change so $\gamma(p).request = \gamma'(p).request$ and $\gamma(q).request = \gamma'(q).request$. As $\gamma'(p).request \not\Rightarrow \gamma'(q).request$, then $\gamma(p).request \not\Rightarrow \gamma(q).request$.

   Let study the other cases:

   (a) $\gamma(p).s = IN$: If $\gamma(q).s = IN$ then p is not locally correct in γ. It is a contradiction. If $\gamma(q).s = WIN$ then the only action that q can make to jump to the state IN is $Enter$, but this action is enabled only if $\nexists n \in \mathcal{N}_q, \gamma(n).s = IN \wedge \gamma(p).request \not\Rightarrow \gamma(n).request$. It is a contradiction.

   (b) $\gamma(p).s = WIN$: If $\gamma(q).s = IN$ then p is not locally correct in γ. It is a contradiction. If $\gamma(q).s = WIN$, q can only make the action $Enter$ to jump into the state IN, but this action is enabled only if $\nexists n \in \mathcal{N}_q, \gamma(n).s = WIN \wedge \gamma(p).request \not\Rightarrow \gamma(n).request$. It is a contradiction.

2. $\gamma'(p).s = WIN \wedge \neg\gamma'(p).CompatibleWithIn$.

   Then $\exists q \in \mathcal{N}_p, \gamma'(q).s = IN \wedge \gamma'(p).request \not\Rightarrow \gamma'(q).request$. There is no action to jump in one step from the states $IDLE$ or TRY to IN and there is no action to jump from the states $IDLE$ or IN to WIN in one step. Hence, $\gamma(p).s \neq IDLE \wedge \gamma(p).s \neq IN$ and $\gamma(q).s \neq IDLE \wedge \gamma(q).s \neq TRY$.

   Note that for the same reasons than in case 1, $\gamma(p).request \not\Rightarrow \gamma(q).request$.

   Let study the other cases:

   (a) $\gamma(p).s = TRY$: The only action that p can make to go to the state WIN is $Rand$ but this action is enabled only if $\nexists n \in \mathcal{N}_p, (\gamma(n).s = WIN \vee \gamma(n).s = IN) \wedge \gamma(p).request \not\Rightarrow \gamma(q).request$. This is a contradiction.

   (b) $\gamma(p).s = WIN$: See the case 1b.

Hence, p is locally correct in γ'. □

Lemma 12. *If p is not locally correct in γ and executes an action then p is locally correct in the new configuration γ'.*

Proof. If p is not locally correct in γ, there are two cases:

1. $\gamma(p).s = IN \wedge \neg\gamma(p).CompatibleWithIn$: The only action that can execute p is $RstI$. So $\gamma'(p).s = IDLE$ and p is locally correct in γ'.

2. $\gamma(p).s = WIN \wedge \neg\gamma(p).CompatibleWithIn$: The only action that can execute p is $RstW$. So $\gamma'(p) = TRY$ and p is locally correct in γ'.

□

Definition 7 (Annoying neighbor)**.** In a configuration γ, let p be a process such that $\gamma(p).s = TRY \vee \gamma(p).s = WIN$. A neighbor $q \in \mathcal{N}_p$ of p is said *annoying* for p if $(\gamma(q).s = IN \vee \gamma(q).s = WIN) \wedge \gamma(p).request \neq \gamma(q).request$. Otherwise, q is said *non-annoying* for p. We note $Annoying_p(\gamma)$ the set of annoying neighbors of p in γ.

Lemma 13. *In a legitimate configuration, with a positive probability, a non-annoying neighbor will stay non-annoying after a step.*

Proof. In a configuration γ, if $q \in \mathcal{N}_p$ is not an annoying neighbor for p then there are four cases:

- $\gamma(q).s = IDLE$: q executes Req and goes to the state TRY. Otherwise, q stays in $IDLE$.

- $\gamma(q).s = TRY$: If $\exists x \in \mathcal{N}_q, (\gamma(x).s = IN \vee \gamma(x).s = WIN) \wedge \gamma(q).request \neq \gamma(x).request$, then q stays in the state TRY and only increments its clock, else executing $Rand$, q stays in the state TRY with probability $\mathsf{p} > 0$.

- $\gamma(q).s = WIN \wedge \gamma(p).request \rightleftharpoons \gamma(q).request$: q cannot execute its critical section in one step (because it must go to the state IN first). So its request and the request of p do not change after a step. Then q stays non-annoying.

- $\gamma(q).s = IN \wedge \gamma(p).request \rightleftharpoons \gamma(q).request$: q executes Rel and goes to the state $IDLE$.

□

Lemma 14. *In a legitimate configuration, with a positive probability, an annoying neighbor will become non-annoying after at most two steps.*

Proof. In a configuration γ, if $q \in \mathcal{N}_p$ is an annoying neighbor for p then there are two cases:

- $\gamma(q).s = IN \wedge \gamma(p).request \neq \gamma(q).request$: q executes Rel and jumps to the state $IDLE$. Then, with a positive probability, it stays non-annoying (Lemma 13).

- $\gamma(q).s = WIN \wedge \gamma(p).request \neq \gamma(q).request$: If $\nexists x \in \mathcal{N}_q, \gamma(x).s = WIN \wedge \gamma(q).request \neq \gamma(x).request$, q executes $Enter$, goes to the state IN and then goes to the state $IDLE$ executing Rel. Else, q goes to the state TRY with probability $\mathsf{p} > 0$ executing $Rand$ and stays in this state with probability $\mathsf{p} > 0$ executing $Rand$ another time.

□

Lemma 15. *If a process is requested in a legitimate configuration, it will almost surely execute its critical section*

Proof. Same arguments as in the proof of Lemma 10. □

Theorem 2. ALGO. 3 satisfies the probabilistic snap-stabilizing guaranteed service local resource allocation specification.

Proof. Let $r = (\gamma_i)_{i\geq 0}$ under a daemon d.

- $r \in Safe$: $\forall k \geq 0, \forall p \in V$, if $Result(\gamma_k(p), \gamma_{k+1}(p))$ is true, then there is $\gamma_k(p).s = IN \wedge \gamma_k(p).CompatibleWithIn \wedge \gamma_{k+1}(p).s = IDLE$. Then, by definition, $Computing(\gamma_k(p)) \wedge \forall q \in \mathcal{N}_p, \neg Computing(\gamma_k(q)) \vee (\gamma_k(p).request() \rightleftharpoons \gamma_k(q).request())$.

  So $CorrectResult(\gamma_0 \dots \gamma_k, p)$ is true.

- $r \in Live$ with probability 1:
  - $\forall k \geq 0, \forall p \in V$, if $\gamma_k(p).s \neq IDLE$, p is computing a service and cannot answer to a new request. But, in a finite time, p will almost surely end its current computation and come back to the state $IDLE$ (Lemma 15). Then, if $\gamma_k(p).request() \neq \bot$, after a finite time, p will be selected by the daemon to execute an action in a configuration γ_l (with $l \geq k$). If the application still request the resource, the guard of Req is true ($\gamma_l(p).s = IDLE \wedge \gamma_l(p).request() \neq \bot$) and the result is $\gamma_{l+1}(p).s = TRY$. So $Start(\gamma_l(p), \gamma_{l+1}(p))$ is true.
  - $\forall k \geq 0, \forall p \in V$, if $Start(\gamma_k(p), \gamma_{k+1}(p))$ then $\gamma_{k+1}(p).s = TRY$. Using Lemma 15, in a finite time, p will almost surely compute its critical section, *i.e.* execute Rel. Then, $\exists l > k$ such as the guard of Rel is true ($\gamma_l(p).s = IN \wedge \gamma_l(p).CompatibleWithIn$) and the result of the execution of Rel is $\gamma_{l+1}(p).s = IDLE$. So $Result(\gamma_l(p), \gamma_{l+1}(p))$ is true.

□

6 Second Probabilistic Snap-Stabilizing Algorithm For Local Mutual Exclusion Problem

This algorithm is a parallel composition [13] of: a coloring module (ALGO. 4) and a synchronization module (ALGO. 5).

6.1 The Coloring Module (ALGO. 4)

The algorithm proposed in [3] is a probabilistic self-stabilizing algorithm for the vertex coloring problem. In other words, it assigns colors to the nodes in such a way that two neighbors do not have the same color.

In this algorithm, when a process has the same color as one of its neighbors, it changes its color using the action CCA. To choose its new color, it draws one in the set of colors that are not used by its neighbors plus its own color with uniform probability.

Here we will use $\Delta + 1$ colors where Δ is an upper bound on the degree of the network.

Algorithm 4 Coloring Module [3]

```
Constants
    $\Delta$: an upper bound on the degree of the network
    $\mathcal{C} = \{0, ..., \Delta\}$: set of possible colors
    $\mathcal{C}_p$: set of colors of the neighbors of $p$
Variables
    $c_p \in \mathcal{C}$
Functions
    $random : S$ (a set of colors) $\rightarrow$ a color in $S$ chosen with uniform probability
Predicates
    $ChangeColor_p \equiv \exists q \in \mathcal{N}_p, c_p = c_q$
Actions
    $CCA : ChangeColor_p \rightarrow c_p := random((\mathcal{C} \backslash \mathcal{C}_p) \cup \{c_p\});$
```

6.2 The Synchronization Module (ALGO. 5)

The synchronization module is a slighty updated version of the synchronization scheme used in ALGO. 1. We assume that we have a strict total order $\ll$ on the colors used by ALGO. 4.

In ALGO. 1, several neighbors can have the same clock, can be enabled to execute NA action and can be selected by the daemon. Then, these processes execute a synchronous step. To avoid this, we use the colors of ALGO. 4 to differentiate these processes and simulate a local sequential behavior. When a process executes a GA action, none of its neighbors do simultaneously; it can thus, if requested, executes its critical section.

Algorithm 5 Synchronization Module (Updating Version of [5])

```
Inputs
    $c_p$: from ALGO. 4
    ⟨ critical section ⟩
Constants
    $N$: an upper bound on the size of the network
Variables
    $t_p \in \mathcal{X} = \{-\alpha, \dots, 0, \dots, K-1\}, \quad \alpha \geq N-2, \quad K \geq N$
Functions
    $\varphi : x \to \begin{cases} (x+1) & \text{if } x < 0 \\ (x+1) \pmod{K} & \text{otherwise} \end{cases}$
    $\ll$: strict total order on the colors in $\mathcal{C}$
Predicates
    $Correct_p(q) \equiv (\varphi(t_p) = t_q) \vee (t_p = t_q) \vee (t_p = \varphi(t_q))$
    $UnisonCorrect_p \equiv \forall q \in \mathcal{N}_p, Correct_p(q)$
    $ColoringCorrect_p \equiv \forall q \in \mathcal{N}_p, c_p \neq c_q$
    $CanGo_p \equiv ColoringCorrect_p \wedge (\forall q \in \mathcal{N}_p, (\varphi(t_p) = t_q) \vee (t_p = t_q \wedge c_p \ll c_q))$
    $ConvergenceStep_p \equiv (t_p < 0) \wedge (\forall q \in \mathcal{N}_p, (t_q \leq 0) \wedge (t_p \leq t_q))$
    $ResetInit_p \equiv (t_p > 0) \wedge \neg UnisonCorrect_p$
Actions
    $GA$ :  $CanGo_p$            $\to$  ⟨ critical section ⟩; $t_p := \varphi(t_p)$;
    $CA$ :  $ConvergenceStep_p$  $\to$  $t_p := \varphi(t_p)$;
    $RA$ :  $ResetInit_p$        $\to$  $t_p := -\alpha$;
```

6.3 Synchronization Between the Two Parts of the Algorithm

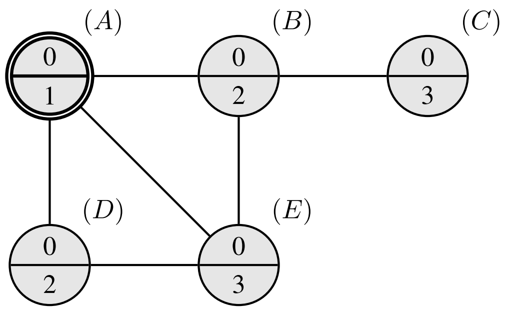
When ALGO. 4 has not converged, the processes execute in parallel the actions of ALGO. 4 and ALGO. 5. Assume that a process p is selected by the daemon. If the guard of RA is true and the guard of CCA is true, p executes these two actions in the same step. If CCA is true and no actions of ALGO. 5 is enabled, p executes only CCA. Finally, if one of the action of ALGO. 5 is enabled but not CCA, p executes only this action.

When ALGO. 4 has converged, its action CCA is (by definition) never enabled. Afterwhile, $ColoringCorrect$ is true and ALGO. 5 plays alone.

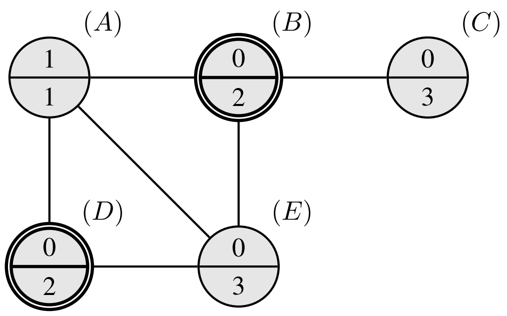
6.4 Example

FIG. 12 is an example of execution of ALGO. 5 after convergence of vertex coloring and unison. The colors are represented by integers and we assume that $\ll$ is the natural order on the integers. The clock is represented in the top of the node and its color is in the bottom. The enabled nodes are circled twice.

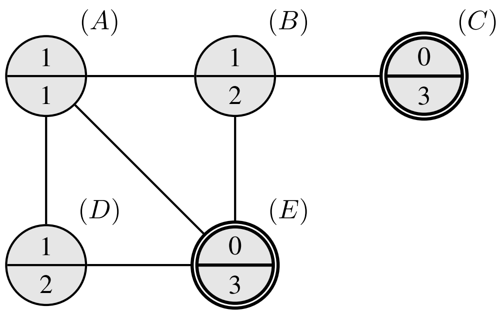
On FIG. (a), (A) is enabled because it has the same clock as its neighbors but its color is the smaller. Even if (C) is not a neighbor of (A), (C) is not enabled because its color is bigger than the one of its neighbor (B) which has the same clock.



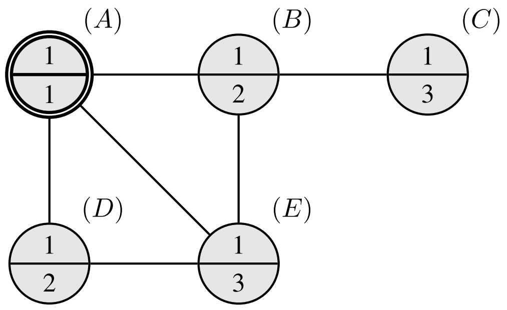
(a) The only process enabled is (A). It is circled twice.



(b) Then (B) and (D) are enabled. As they are not neighbors, they can execute their critical section simultaneously.



(c) Then (C) and (E) are enabled.



(d) Finally, (A) is enabled again.

Figure 12: Example of execution. The top of the process represents its clock, the bottom its color. We assume the natural order on the colors.

After the execution of the critical section and the incrementation of its clock by (A), (B) and (D) are enabled (FIG. (b)). (A) cannot be enabled another time because its clock is bigger than the ones of its neighbors.

On FIG. (c), (C) and (E) are enabled. And finally, (A) is enabled again on FIG. (d).

6.5 Instantiation of the specification by the algorithm

The predicates of the specification are instantiated the following way:

- $Request(s) \equiv s.request()$
- $Start(s, s') \equiv s'.UnisonCorrect() \wedge s'.ColoringCorrect()$
- $Result(s, s') \equiv \varphi(s.t) = s'.t$
- $Computing(s) \equiv s.CanGo$

6.6 Proof

Assume an execution of the system under a daemon d, where the requests are set and with any initial configuration γ_0.

Definition 8 (Legitimate configuration of the synchronization module)**.** A configuration γ is *legitimate for the synchronization module* if and only if $\forall p \in V, \gamma(p).UnisonCorrect$.

Lemma 16. *If the system is in an illegitimate configuration of the synchronization module, it converges in a finite time to a legitimate configuration.*

Proof. The correction actions CA and RA are identical to the original algorithm. In a finite number of actions CA and RA, the system comes back to a legitimate state of the unison [9]. So, if the system is an illegitimate configuration of the synchronization module, it converges in a finite time to a legitimate configuration and the convergence time is identical to the one of [5]. □

Definition 9 (Legitimate configuration)**.** A configuration γ of the system is *legitimate* if and only if $\forall p \in V, \gamma(p).UnisonCorrect \land \gamma(p).ColoringCorrect$. Otherwise γ is illegitimate.

Lemma 17. *In every configuration of a run $r = (\gamma_i)_{i \geq 0}$, at least one process is enabled.*

Proof. As the coloring module is self-stabilizing [3] and the synchronization module converges in a finite time (Lemma 16), there is no deadlock in an illegitimate configuration.

Let γ a legitimate configuration. Assume that there is no process enabled in γ. Let X be the set of processes that are not enabled in γ but would be enabled in the original algorithm of unison [5]. $X \neq \emptyset$ since unison has no deadlock. Let $q_0 \in X$. Then $\exists q_1 \in \mathcal{N}_{q_0}, \gamma(q_0).t = \varphi(\gamma(q_1).t) \lor (\gamma(q_0).t = \gamma(q_1).t \land \gamma(q_1).c \ll \gamma(q_0).c)$. By assumption, q_1 is not enabled. So $\exists q_2 \in \mathcal{N}_{q_1}, \gamma(q_1).t = \varphi(\gamma(q_2)).t \lor (\gamma(q_1).t = \gamma(q_2).t \land \gamma(q_2).c \ll \gamma(q_1).c)$ *etc.*

This way, we construct a sequence $(q_i)_{i \geq 0}$ of process such that q_i is the process that "prevents" q_{i-1} of being enabled. As $\gamma(q_i).t = \varphi(\gamma(q_{i+1}).t) \lor \gamma(q_{i+1}).c \ll \gamma(q_i).c$, $q_i = q_j$ if and only if $i = j$. Then $| (q_i)_{i \geq 0} | \leq n$. So the last process of the sequence is enabled. It is a contradiction. □

Lemma 18. *The set of legitimate configurations is closed.*

Proof. Let γ be a legitimate configuration and γ' a configuration reachable from γ in one step.

The coloring module is self-stabilizing [3] so $\forall p \in V, \gamma'(p).ColoringCorrect$. The unison is self-stabilizing [5] and the only difference between NA of the original algorithm and GA is that the guard of GA has an additional constraint. There are no consequences on the self-stabilization of the algorithm. So $\forall p \in V, \gamma'(p).UnisonCorrect$. □

Lemma 19. *In a run $(\gamma_i)_{i \geq 0}$ where γ_0 is legitimate, two neighbors cannot execute their critical section concurrently.*

Proof. Using Lemma 18, if γ_0 is legitimate then $\forall i \geq 0$, γ_i is legitimate. The only action in which a process executes the critical section is GA.

Let $p, q \in V$ two neighbors ($q \in \mathcal{N}_p$). Assume that there is a configuration γ_i such that p and q evaluate the guard of GA to true. If p evaluates the guard of GA to true then $\forall n \in \mathcal{N}_p, \varphi(\gamma_i(p).t) = \gamma_i(n).t \lor (\gamma_i(p).t = \gamma_i(n).t \land \gamma_i(p).c \ll \gamma_i(n).c)$. In particular, $\varphi(\gamma_i(p).t) = \gamma_i(q).t \lor ((\gamma_i(p).t = \gamma_i(q).t \land \gamma_i(p).c \ll \gamma_i(q).c)$. With the same argument on q, we have $\varphi(\gamma_i(q).t) = \gamma_i(p).t \lor ((\gamma_i(q).t = \gamma_i(p).t \land \gamma_i(q).c \ll \gamma_i(p).c)$.

Let examine the different cases:

- If $\varphi(\gamma_i(p).t) = \gamma_i(q).t$ then $\gamma_i(p) \neq \gamma_i(q).t$ and $\gamma_i(p).t \neq \varphi(\gamma_i(q).t)$. It is a contradiction.
- If $\gamma_i(p).t = \gamma_i(q).t \land \gamma_i(p).c \ll \gamma_i(q).c$, $\gamma_i(q).c \not\ll \gamma_i(p).c$ because $\ll$ is a strict total order on the colors and γ_i is a legitimate configuration so $\gamma_i(p).c \neq \gamma_i(q).c$. It is a contradiction.

□

Lemma 20. *A system in an illegitimate configuration converges to a legitimate configuration in expected $O(\Delta n)$ rounds where Δ is the degree of the system.*

Proof. The coloring module converges in expected $O(\Delta n)$ rounds [3]. The unison converges in $O(n)$ rounds [4]. As the two modules converge in parallel, the system converges in the time of the longer, *i.e.* the system converges in expected $O(\Delta n)$ rounds. □

Lemma 21. *A system in an illegitimate configuration converges to a legitimate configuration in expected $O(\mathcal{D}n^3)$ steps where $\mathcal{D}$ is the diameter of the system.*

Proof. The coloring module converges in expected $O(\Delta n)$ steps [3] (where Δ is the degree of the network). The unison converges in $O(\mathcal{D}n^3)$ steps [9]. As the two modules converge in parallel, the system converges in the time of the longer, *i.e.* the system converges in expected $O(\mathcal{D}n^3)$ steps. □

Lemma 22. *In a run $r = (\gamma_i)_{i \geq 0}$ where γ_0 is legitimate, a process p will execute its critical section in a finite time.*

Proof. Thanks to the unison and the coloring, there is a total order on the neighbors. Indeed, it is possible to sort a process and its neighbors by the value of their clocks. Then the processes which has the same clock value are sorted by their color using $\ll$.

Let $p \in V$. Assume that p never executes its critical section in r. Then $\exists q \in \mathcal{N}_p$ such that q executes infinitely often its critical section.

If q executes its critical section in γ_k then $\forall n \in \mathcal{N}_q, (\varphi(\gamma_k(q).t) = \gamma_k(n).t) \vee (\gamma_k(q).t = \gamma_k(n).t \wedge \gamma_k(q).c \ll \gamma_k(n).c)$. In particular, it is true for $n = p$. Let examine the two cases:

- $\varphi(\gamma_k(q).t) = \gamma_k(p).t$: Then, $\gamma_{k+1}(q).t = \gamma_{k+1}(p).t$. If $\gamma_{k+1}(q).c \not\ll \gamma_{k+1}(p).c$ then q cannot executes its critical section until p increments its clock and p can only increment its clock by executing GA and so by executing its critical section (contradiction). Otherwise, after the next time that q executes its critical section (let this configuration be $\gamma_{k'}$), $\gamma_{k'}(q).t = \varphi(\gamma_{k'}(p).t)$. Then q cannot execute its critical section until p increments its clock (contradiction).

- $\gamma_k(q).t = \gamma_k(p).t \wedge \gamma_k(q).c \ll \gamma_k(p).c$: Then $\gamma_{k+1}(q).t = \varphi(\gamma_{k+1}(p).t)$ so q cannot execute its critical section until p increments its clock (contradiction).

□

Theorem 3. The algorithm satisfies the probabilistic snap-stabilizing guaranteed service local mutual exclusion specification.

Proof. Let a run $r = (\gamma_i)_{i \geq 0}$ under a daemon d.

- $r \in Safe$: $\forall k \geq 0$, $\forall p \in V$, if $\exists l < k, Start(\gamma_l(p), \gamma_{l+1}(p))$ then by Lemma 18, $\forall i \geq l + 1, \gamma_i(p).UnisonCorrect() \wedge \gamma_i(p).ColoringCorrect()$.

  In particular, $\gamma_k(p).UnisonCorrect() \wedge \gamma_k(p).ColoringCorrect()$. Then the only action that p can execute to increment its clock when the unison and the coloring have converged is GA. So, if $Result(\gamma_k(p), \gamma_{k+1}(p))$, then $\gamma_k(p).CanGo$. By Lemma 19, two neighbors cannot concurrently execute GA so $\forall q \in \mathcal{N}_p$, $\neg\gamma_k(q).CanGo$. In conclusion, $CorrectResult(\gamma_0 \ldots \gamma_k, p)$ is true.

- $r \in Live$:
  - $\forall k \geq 0, \forall p \in V$, in a finite time, the unison and the coloring converge (Lemma 20). Hence, $\exists l > k, Request(\gamma_l(p)) \Rightarrow Start(\gamma_l(p), \gamma_{l+1}(p))$.
  - $\forall k \geq 0, \forall p \in V$, if $Start(\gamma_k(p), \gamma_{k+1}(p))$ then γ_{k+1} is a legitimate configuration. Hence, by Lemma 22, in a finite time p will execute its critical section by executing GA (and increments its clock) so $\exists l > k, \varphi(\gamma_l(p).t) = \gamma_{l+1}(p).t$.

□

7 Generalization To Some Other Local Resource Allocation Problems

We introduce a generalization of this algorithm to the local resource allocation problems expressible with the relation $\rightleftharpoons$.

The coloring module is identical to ALGO. 4. In the synchronization module (ALGO. 6), each process has an additional variable $request$ which expresses the resource required. A process can execute its critical section if its clock is the smaller in its neighborhood and if its color is smaller or its request compatible with the ones of its neighbors which have the same clock.

Algorithm 6 Generalization of the Synchronization Module

```
Inputs
    $c_p$: from ALGO. 4
    $\langle$ critical section $\rangle$
    $request_p$: from the application
Constants
    $N$: an upper bound on the size of the network
Variables
    $t_p \in \mathcal{X} = \{-\alpha, \ldots, 0, \ldots, K-1\}$,   $\alpha \geq N-2$,   $K \geq N$
Functions
    $\varphi : x \rightarrow \begin{cases} (x+1) & \text{if } x < 0 \\ (x+1) \pmod{K} & \text{otherwise} \end{cases}$
    $\ll$: total order on the colors in $\mathcal{C}$
    $\rightleftharpoons$: resource compatibility
Predicates
    $Correct_p(q)$            $\equiv$  $(\varphi(t_p) = t_q) \vee (t_p = t_q) \vee (t_p = \varphi(t_q))$
    $UnisonCorrect_p$         $\equiv$  $\forall q \in \mathcal{N}_p, Correct_p(q)$
    $ColoringCorrect_p$       $\equiv$  $\forall q \in \mathcal{N}_p, c_p \neq c_q$
    $CanGo_p$                 $\equiv$  $ColoringCorrect_p \wedge (\forall q \in \mathcal{N}_p, (\varphi(t_p) = t_q) \vee (t_p = t_q \wedge$
                                            $(c_p \ll c_q \vee request_p \rightleftharpoons request_q)))$
    $ConvergenceStep_p$       $\equiv$  $(t_p < 0) \wedge (\forall q \in \mathcal{N}_p, (t_q \leq 0) \wedge (t_p \leq t_q))$
    $ResetInit_p$             $\equiv$  $(t_p > 0) \wedge \neg UnisonCorrect_p$
Actions
    $GA$ :  $CanGo_p$             $\rightarrow$  $\langle$ critical section $\rangle$; $request_p := \bot$; $t_p := \varphi(t_p)$;
    $CA$ :  $ConvergenceStep_p$   $\rightarrow$  $t_p := \varphi(t_p)$;
    $RA$ :  $ResetInit_p$         $\rightarrow$  $t_p := -\alpha$;
```

7.1 Instantiation of the specification by the algorithm

The predicates of the specification are instantiated the following way:

- $Request(s) \equiv s.request()$
- $Start(s, s') \equiv s'.UnisonCorrect() \wedge s'.ColoringCorrect()$
- $Result(s, s') \equiv \varphi(s.t) = s'.t$
- $Computing(s) \equiv \gamma_k(p).CanGo$

7.2 Proof

We go back over the proofs of lemmas that change due to the generalization. The others definitions, lemmas or proofs stay identical.

Lemma 23. *In every configuration of a run $r = (\gamma_i)_{i \geq 0}$, at least one process is enabled.*

Proof. As the coloring module is self-stabilizing [3] and the synchronization module converges in a finite time (Lemma 16), there is no deadlock in an illegitimate configuration.

Let γ a legitimate configuration. Assume that there is no processes enabled in γ. Let X be the set of process that are not enabled in γ but would be enabled in the original algorithm of unison [5]. $X \neq \emptyset$ since unison has no deadlocks. Let $q_0 \in X$. Then $\exists q_1 \in \mathcal{N}_{q_0}, \gamma(q_0).t = \varphi(\gamma(q_1).t) \vee (\gamma(q_0).t = \gamma(q_1).t \wedge \gamma(q_1).c \ll \gamma(q_0).c \wedge \gamma(q_0).request \not\rightleftharpoons \gamma(q_1).request)$. By assumption, q_1 is not enabled. So $\exists q_2 \in \mathcal{N}_{q_1}, \gamma(q_1).t = \varphi(\gamma(q_2)).t \vee (\gamma(q_1).t = \gamma(q_2).t \wedge \gamma(q_2).c \ll \gamma(q_1).c \wedge \gamma(q_1).request \not\rightleftharpoons \gamma(q_2).request)$ *etc.*

This way, we construct a sequence $(q_i)_{i \geq 0}$ of process such that q_i is the process that "prevents" q_{i-1} of being enabled. As $\gamma(q_i).t = \varphi(\gamma(q_{i+1}).t) \vee \gamma(q_{i+1}).c \ll \gamma(q_i).c$, $q_i = q_j$ if and only if $i = j$. Then $\mid (q_i)_{i \geq 0} \mid \leq n$. So the last process of the sequence is enabled. It is a contradiction. □

Lemma 24. *In a run $(\gamma_i)_{i\geq 0}$ where γ_0 is legitimate, two neighbors cannot executes their critical section concurrently if their requests are conflicting.*

Proof. Using Lemma 18, if γ_0 is legitimate then $\forall i \geq 0$, γ_i is legitimate. The only action in which a process executes the critical section is GA.

Let $p, q \in V$ two neighbors ($q \in \mathcal{N}_p$). Assume that there is a configuration γ_i such that p and q evaluate the guard of GA to true and $\gamma_i(p).request \not\rightleftharpoons \gamma_i(q).request$. If p evaluates the guard of GA to true then $\forall n \in \mathcal{N}_p, \varphi(\gamma_i(p).t) = \gamma_i(n).t \vee (\gamma_i(p).t = \gamma_i(n).t \wedge (\gamma_i(p).c \ll \gamma_i(n).c \vee \gamma_i(p).request \rightleftharpoons \gamma_i(n).request))$. In particular, $\varphi(\gamma_i(p).t) = \gamma_i(q).t \vee ((\gamma_i(p).t = \gamma_i(q).t \wedge (\gamma_i(p).c \ll \gamma_i(q).c \vee \gamma_i(p).request \rightleftharpoons \gamma_i(q).request))$. With the same argument on q, we have $\varphi(\gamma_i(q).t) = \gamma_i(p).t \vee ((\gamma_i(q).t = \gamma_i(p).t \wedge (\gamma_i(q).c \ll \gamma_i(p).c \vee \gamma_i(q).request \rightleftharpoons \gamma_i(p).request))$.

Let examine the different cases:

- If $\varphi(\gamma_i(p).t) = \gamma_i(q).t$ then $\gamma_i(p) \neq \gamma_i(q).t$ and $\gamma_i(p).t \neq \varphi(\gamma_i(q).t)$. It is a contradiction.
- By assumption, $\gamma_i(p).t = \gamma_i(q).t \wedge \gamma_i(p).request \rightleftharpoons \gamma_i(q).request$ is not possible.
- If $\gamma_i(p).t = \gamma_i(q).t \wedge \gamma_i(p).request \not\rightleftharpoons \gamma_i(q).request \wedge \gamma_i(p).c \ll \gamma_i(q).c$, $\gamma_i(q).c \not\ll \gamma_i(p).c$, because $\ll$ is a total order on the colors and γ_i is a legitimate configuration so $\gamma_i(p).c \neq \gamma_i(q).c$, and $\gamma_i(q).request \not\rightleftharpoons \gamma_i(p).request$ by assumption. It is a contradiction.

□

Lemma 25. *In a run $r = (\gamma_i)_{i\geq 0}$ where γ_0 is legitimate, a process p will execute its critical section in a finite time.*

Proof. Thanks to the unison and the coloring, there is a total order on the neighbors. Indeed, it is possible to sort a process and its neighbors by the value of their clocks. Then the processes which have the same clock value are sorted using $\ll$.

Let $p \in V$. Assume that p never executes its critical section in r. Then $\exists q \in \mathcal{N}_p$ such that q executes infinitely often its critical section.

If q executes its critical section in γ_k then $\forall n \in \mathcal{N}_q, (\varphi(\gamma_k(q).t) = \gamma_k(n).t) \vee (\gamma_k(q).t = \gamma_k(n).t \wedge (\gamma_k(q).c \ll \gamma_k(n).c \vee \gamma_k(q).request \rightleftharpoons \gamma_k(n).request))$. In particular, it is true for $n = p$. Let examine the two cases:

- $\varphi(\gamma_k(q).t) = \gamma_k(p).t$: Then, $\gamma_{k+1}(q).t = \gamma_{k+1}(p).t$. If $\gamma_{k+1}(q).c \not\ll \gamma_{k+1}(p).c \wedge \gamma_{k+1}(q).request \not\rightleftharpoons \gamma_{k+1}(p).request$ then q cannot executes its critical section until p increments its clock and p can only increment its clock by executing GA and so by executing its critical section (contradiction). Otherwise, after the next time that q executes its critical section (let this configuration be $\gamma_{k'}$), $\gamma_{k'}(q).t = \varphi(\gamma_{k'}(p).t)$. Then q cannot execute its critical section until p increments its clock (contradiction).
- $\gamma_k(q).t = \gamma_k(p).t \wedge \gamma_k(q).c \ll \gamma_k(p).c$: Then $\gamma_{k+1}(q).t = \varphi(\gamma_{k+1}(p).t)$ so q cannot execute its critical section until p increments its clock (contradiction).

□

Theorem 4. The algorithm satisfies the probabilistic snap-stabilizing guaranteed service local resource allocation specification.

Proof. The proof is similar to the one of Theorem 3. □

8 Conclusion

We have proposed two probabilistic snap-stabilizing algorithms to solve the local mutual exclusion problem and their generalizations to some other local resource allocation problems *e.g.* local readers-writers problem *etc.* in any anonymous network. These algorithms are correct under a distributed unfair daemon.

In the first one, each time a process requests a resource, it must compete with its neighbors to determine which one (or ones according to the problem) will be served first. In the second one, a synchronization scheme and a coloring are used to simulate a local sequential execution.

The perspectives of our work is the generalization of these algorithms to local resource allocation problems that are not expressible with the relation of compatibility *e.g.* local l-exclusion. Such kind of problems needs a vision at distance 2 of the configuration of the network. This would require heavy mechanisms and would not fit in the scheme of our solutions.

To study if these algorithms satisfy the maximal concurrency, *i.e.* if they allows maximum number of processes to concurrently execute their critical section, would be interesting. This property allows us, for example, to dismiss algorithms which solve the mutual exclusion problem (where only one process can execute its critical section at a given time) as a solution of the local mutual exclusion problem (where two processes can concurrently execute their critical section if they are not neighbors).

References

[1] Karine Altisen and Stéphane Devismes. On probabilistic snap-stabilization. Private communication, 2013. 1, 1

[2] Karine Altisen, Stéphane Devismes, and Anaïs Durand. Private communication, 2013. 4.6, 4.6

[3] Samuel Bernard, Stéphane Devismes, Katy Paroux, Maria Potop-Butucaru, and Sébastien Tixeuil. Probabilistic self-stabilizing vertex coloring in unidirectional anonymous networks. In *ICDCN*, pages 167–177, 2010. 1, 6.1, 4, 6.6, 6.6, 6.6, 6.6, 7.2

[4] Christian Boulinier. *L'unisson*. PhD thesis, Université de Picardie Jules Verne, 2007. 4.6, 6.6

[5] Christian Boulinier, Franck Petit, and Vincent Villain. When graph theory helps self-stabilization. In *PODC*, pages 150–159, 2004. 4.1, 1, 4.6, 5, 6.6, 6.6, 6.6, 7.2

[6] Alain Bui, Ajoy K. Datta, Franck Petit, and Vincent Villain. State-optimal snap-stabilizing pif in tree networks. In *In Proceedings of the Fourth Workshop on Self-Stabilizing Systems*, pages 78–85. IEEE Computer Society Press, 1999. 1

[7] Sébastien Cantarell, Ajoy Kumar Datta, and Franck Petit. Self-stabilizing atomicity refinement allowing neighborhood concurrency. In *Self-Stabilizing Systems*, pages 102–112, 2003. 1, 2.4

[8] Stéphane Devismes. Cours d'algorithmique répartie, RICM5, 2012. 1

[9] Stéphane Devismes and Franck Petit. On efficiency of unison. In *TADDS*, pages 20–25, 2012. 4.6, 6.6, 6.6

[10] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974. 1

[11] Edsger W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1(1):5–6, 1986. 1

[12] Hirotsugu Kakugawa and Masafumi Yamashita. Self-stabilizing local mutual exclusion on networks in which process identifiers are not distinct. In *SRDS*, pages 202–211, 2002. 1

[13] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2nd edition, 2001. 6