Concurrency in Snap-Stabilizing Local Resource Allocation

Karine Altisen, Stéphane Devismes, Anaïs Durand

Concurrency in Snap-Stabilizing Local Resource Allocation*

Karine Altisen, Stéphane Devismes, and Anaïs Durand

VERIMAG UMR 5104
Université Grenoble Alpes, France
firstname.lastname@imag.fr

Abstract

In distributed systems, resource allocation consists in managing fair access of a large number of processes to a typically small number of reusable resources. As soon as the number of available resources is greater than one, the efficiency in concurrent accesses becomes an important issue, as a crucial goal is to maximize the utilization rate of resources. In this paper, we tackle the concurrency issue in resource allocation problems. We first characterize the maximal level of concurrency we can obtain in such problems by proposing the notion of *maximal-concurrency*. Then, we focus on Local Resource Allocation problems (LRA). Our results are both negative and positive. On the negative side, we show that it is impossible to obtain maximal-concurrency in LRA without compromising the fairness. On the positive side, we propose a snap-stabilizing LRA algorithm which achieves a high (but not maximal) level of concurrency, called here *strong partial maximal-concurrency*.

*This work has been partially supported by the ANR Persyval Project DACRAW.

Contents

1 Introduction

Mutual exclusion [14, 25] is a fundamental resource allocation problem, which consists in managing fair access of all (requesting) processes to a unique non-shareable reusable resource. This problem is inherently sequential, as no two processes should access this resource concurrently. There are many other resource allocation problems which, in contrast, allow several resources to be accessed simultaneously. In those problems, parallelism on access to resources may be restricted by some of the following conditions:

1. The maximum number of resources that can be used concurrently, *e.g.*, the ℓ-*exclusion* problem [19] is a generalization of the mutual exclusion problem which allows use of ℓ identical copies of a non-shareable reusable resource among all processes, instead of only one, as standard mutual exclusion.

2. The maximum number of resources a process can use simultaneously, *e.g.*, the k-*out-of-*ℓ-*exclusion* problem [27] is a generalization of ℓ-exclusion where a process can request for up to k resources simultaneously.

3. Some topological constraints, *e.g.*, in the *dining philosophers* problem [16], two neighbors cannot use their common resource simultaneously.

For efficiency purposes, algorithms solving such problems must be as parallel as possible. As a consequence, these algorithms should be, in particular, evaluated at the light of the level of concurrency they permit, and this level of concurrency should be captured by a dedicated property. However, most of the solutions to resource allocation problems simply do not consider the concurrency issue, *e.g.*, [4, 6, 8, 20, 22, 24, 26]

Now, as quoted by Fischer *et al.* [19], specifying resource allocation problems without including a property of concurrency may lead to degenerated solutions, *e.g.*, any mutual exclusion algorithm realizes safety and fairness of ℓ-exclusion. To address this issue, Fischer *et al.* [19] proposed an *ad hoc* property to capture concurrency in ℓ-exclusion. This property is called *avoiding* ℓ-*deadlock* and is informally defined as follows: "if fewer than ℓ processes are executing their critical section,[1] then it is possible for another process to enter its critical section, even though no process leaves its critical section in the meantime." Some other properties, inspired from the avoiding ℓ-deadlock property, have been proposed to capture the level of concurrency in other resource allocation problems, *e.g.*, k-out-of-ℓ-exclusion [10] and committee coordination [5]. However, until now, all existing properties of concurrency are specific to a particular problem.

In this paper, we first propose to generalize the definition of avoiding ℓ-deadlock to any resource allocation problems. We call this new property the *maximal-concurrency*. Then, we

[1] The *critical section* is the code that manages the access of a process to its allocated resources.

consider the maximal-concurrency in the context of the *Local Resource Allocation (LRA)* problem, defined by Cantarell *et al.* [8]. LRA is a generalization of resource allocation problems in which resources are shared among neighboring processes. Dining philosophers, local reader-writers, local mutual exclusion, and local group mutual exclusion are particular instances of LRA. In contrast, local ℓ-exclusion and local k-out-of-ℓ-exclusion cannot be expressed with LRA although they also deal with neighboring resource sharing.

Now, we show that algorithms for a wide class of instances of this important problem cannot achieve maximal-concurrency. This impossibility result is mainly due to the fact that fairness of LRA and maximal-concurrency are incompatible properties: it is impossible to implement an algorithm achieving both properties. As unfair resource allocation algorithms are clearly unpractical, we propose to weaken the property of maximal-concurrency. We call *partial maximal-concurrency* this weaker version of maximal concurrency. The goal of *partial maximal-concurrency* is to capture the maximal level of concurrency that can be obtained in LRA without compromising fairness.

We propose a LRA algorithm achieving (strong) partial maximal-concurrency in bidirectional identified networks of arbitrary topology. As additional feature, this algorithm is *snap-stabilizing* [7]. *Snap-stabilization* is a versatile property which enables a distributed system to efficiently withstand transient faults. Informally, after transient faults cease, a snap-stabilizing algorithm *immediately* resumes correct behavior, without external intervention. More precisely, a snap-stabilizing algorithm guarantees that any computation started after the faults cease will operate correctly. However, we have no guarantees for those executed all or a part during faults. By definition, snap-stabilization is a strengthened form of *self-stabilization* [15]: after transient faults cease, a self-stabilizing algorithm *eventually* resume correct behavior, without external intervention.

There exist many algorithms for particular instances of the LRA problem. Many of these solutions have been proven to be self-stabilizing, *e.g.*, [4, 6, 8, 20, 22, 24, 26]. In [6], Boulinier *et al.* propose a self-stabilizing unison algorithm which allows to solve local mutual exclusion, local group mutual exclusion, and local reader-writers problem. There are also many self-stabilizing algorithms for local mutual exclusion [4, 20, 24, 26]. In [22], Huang proposes a self-stabilizing algorithm solving the dining philosophers problem. A self-stabilizing drinking philosophers algorithm is given in [26]. In [8], Cantarell *et al.* generalize the above problems by introducing the LRA problem. They also propose a self-stabilizing algorithm for that problem. To the best of our knowledge, no other paper deals with the general instance of LRA and no paper proposes snap-stabilizing solution for any particular instance of LRA. Finally, none of the aforementioned papers (especially [8]) consider the concurrency issue. Finally, note that there exist weaker versions of the LRA problem, such as the (local) *conflict managers* proposed in [21] where the fairness is replaced by a progress property.

Roadmap. The next section introduce the computation model and the specification of the LRA problem. In Section 3, we introduce the property of maximal-concurrency, show the impossibility result, and introduce the property of partial maximal-concurrency. Our algorithm is presented in Section 6. We prove its correctness in Section 7 and its partial maximal-concurrency in Section 8. We conclude in Section 9.

2 Computational Model and Specifications

2.1 Distributed Systems

We consider *distributed systems* composed of n *processes*. A process p can (directly) communicate with a subset $\mathcal{N}_p$ of other processes, called its *neighbors*. These communications are assumed to be *bidirectional*, *i.e.*, for any two processes p and q, $q \in \mathcal{N}_p$ if and only if $p \in \mathcal{N}_q$. Hence, the topology of the network can be modeled by a simple undirected graph $G = (V, E)$, where V is the set of processes and E is the set of edges representing (direct) communication relations. Moreover, we assume that each process has a unique ID encoded as a natural integer. By abuse of notation, we identify the process with its own ID, whenever convenient.

2.2 Locally Shared Memory Model

We consider the *locally shared memory model* in which processes communicate using a finite number of locally shared registers, called *variables*. Each process can read its own variables and those of its neighbors, but can only write to its own variables. The *state* of a process is the vector of values of all its variables. A *configuration* γ of the system is the vector of states of all processes. We denote by $\mathcal{C}$ the set of possible configurations and $\gamma(p)$ the state of process p in configuration γ.

A *distributed algorithm* consists of one *program* per process. The program of a process p is composed of a finite number of *actions*, where each action has the following form:

$$(\langle \text{priority} \rangle) \quad \langle \text{label} \rangle : \langle \text{guard} \rangle \rightarrow \langle \text{statement} \rangle$$

The *labels* are used to identify actions. The *guard* of an action in the program of process p is a Boolean expression involving the variables of p and its neighbors. *Priorities* are used to simplify the guards of the actions. The actual guard of an action "$(j)\ L\ :\ G\ \rightarrow\ S$" at p is the conjunction of G and the negation of the disjunction of all guards of actions at p with priority $i < j$. An action of priority i is said to be of *higher priority* than any action of priority $j > i$. If the actual guard of some action evaluates to true, then the action is said to be *enabled* at p. By definition, a process p is not enabled to execute any (lower priority) action if it is enabled

to execute an action of higher priority. If at least one action is enabled at p, p is also said to be enabled. We denote by $Enabled(\gamma)$ the set of processes enabled in configuration γ. The *statement* of an action is a sequence of assignments on the variables of p. An action can be executed only if it is enabled. In this case, the execution of the action consists in executing its statement.

The asynchronism of the system is materialized by an adversary, called the *daemon*. In a configuration γ, if there is at least one enabled process (*i.e.*, $Enabled(\gamma) \neq \emptyset$), then the daemon selects a non empty subset S of $Enabled(\gamma)$ to perform an *(atomic) step*: Each process of S atomically executes one of its enabled action in γ, leading the system to a new configuration γ'. We denote by $\mapsto$ the relation between configurations such that $\gamma \mapsto \gamma'$ if and only if γ' can be reached from γ in one (atomic) step. An *execution* is a maximal sequence of configurations $\gamma_0, \gamma_1, \ldots$ such that $\forall i > 0$, $\gamma_{i-1} \mapsto \gamma_i$. The term "maximal" means that the execution is either infinite, or ends at a *terminal* configuration γ in which $Enabled(\gamma)$ is empty.

In this paper, we assume a *distributed weakly fair* daemon. "Distributed" means that while the configuration is not terminal, the daemon should select at least one enabled process, maybe more. "Weakly fair" means that there is no infinite suffix of execution in which a process p is continuously enabled without ever being selected by the daemon.

To measure the time complexity of an algorithm, we use the notion of *round*. This latter allows to highlight the execution time according to the speed of the slowest process. The first round of an execution $e = (\gamma_i)_{i \geq 0}$ is the minimal prefix e' of e such that every enabled process in γ_0 either executes an action or is *neutralized* (defined below). Let γ_j be the last configuration of e', the second round of e is the first round of $e'' = (\gamma_i)_{i \geq j}$, and so forth.

Neutralized means that a process p is enabled in a configuration γ_i and is not enabled in γ_{i+1} but does not execute any action during the step $\gamma_i \mapsto \gamma_{i+1}$.

2.3 Snap-Stabilizing Local Resource Allocation

In resource allocation problems, a typically small amount of reusable *resources* is shared among a large number of processes. A process may spontaneously request for one or several resources. When granted, the access to the requested resource(s) is done using a special section of code, called *critical section*. The process can only hold resources for a finite time: eventually, it should release these resources to the system, in order to make them available for other requesting processes. In particular, this means that the critical section is always assumed to be finite. In the following, we denote by $\mathcal{R}_p$ the set of resources that can be accessed by a process p.

2.3.1 Local Resource Allocation

The *Local Resource Allocation (LRA)* problem [8] is based on the notion of compatibility: two resources X and Y are said to be *compatible* if two neighbors can concurrently access them. Otherwise, X and Y are said to be *conflicting*. In the following, we denote by $X \rightleftharpoons Y$ (resp. $X \not\rightleftharpoons Y$) the fact that X and Y are compatible (resp. conflicting). Notice that $\rightleftharpoons$ is a symmetric relation.

Using the compatibility relation, the *local resource allocation* problem consists in ensuring that every process which requires a resource r eventually accesses r while no other conflicting resource is currently used by a neighbor. Notice that the case where there are no conflicting resources is trivial: a process can always use a resource whatever the state of its neighbors. So, from now on, we will always assume that there exists at least one conflict, *i.e.*, there are (at least) two neighbors p, q and two resources X, Y such that $X \in \mathcal{R}_p, Y \in \mathcal{R}_q$ and $X \not\rightleftharpoons Y$. This means, in particular, that any network, considered from now on, contains at least two nodes.

Specifying the relation $\rightleftharpoons$, it is possible to define some classic resource allocation problems in which the resources are shared among neighboring processes.

Example 1: Local Mutual Exclusion. In the *local mutual exclusion* problem, no two neighbors can concurrently access the unique resource. So there is only one resource X common to all processes and $X \not\rightleftharpoons X$.

Example 2: Local Readers-Writers. In the *local readers-writers* problem, the processes can access a file in two different modes: a read access (the process is said to be a *reader*) or a write access (the process is said to be a *writer*). A writer must access the file in local mutual exclusion, while several reading neighbors can concurrently access the file. We represent these two access modes by two resources at every process: R for a "read access" and W for a "write access." Then, $R \rightleftharpoons R$, but $W \not\rightleftharpoons R$ and $W \not\rightleftharpoons W$.

Example 3: Local Group Mutual Exclusion. In the *local group mutual exclusion* problem, there are several resources $r_0, r_1, \ldots, r_k$ shared between the processes. Two neighbors can access concurrently the same resource but cannot access different resources at the same time. Then:

$$\forall i \in \{0,\ldots,k\}, \forall j \in \{0,\ldots,k\}, \begin{cases} r_i \rightleftharpoons r_j & \text{if } i = j \\ r_i \not\rightleftharpoons r_j & \text{otherwise} \end{cases}$$

2.3.2 Snap-Stabilization

Let $\mathcal{A}$ be a distributed algorithm. A *specification* SP is a predicate over all executions of $\mathcal{A}$. In [7], snap-stabilization has been defined as follows: $\mathcal{A}$ is *snap-stabilizing w.r.t.* SP if starting from any arbitrary configuration, all its executions satisfy SP.

Of course, not all specifications — in particular their safety part — can be satisfied when considering a system which can start from an arbitrary configuration. Actually, snap-stabilization's notion of safety is *user-centric*: when the user initiates a computation, then the computed result should be correct. So, we express a problem using a *guaranteed service specification* [2]. Such a specification consists in specifying three properties related to the computation start, computation end, and correctness of the delivered result. (In the context of LRA, this latter property will be referred to as "resource conflict freedom.")

To formally define the guaranteed service specification of the local resource allocation problem, we need to introduce the following four predicates, where p is a process, r is a resource, and $e = (\gamma_i)_{i\geq 0}$ is an execution:

- $Request(\gamma_i, p, r)$ means that an application at p requires r in configuration γ_i. We assume that if $Request(\gamma_i, p, r)$ holds, it continuously holds until p accesses r.
- $Start(\gamma_i, \gamma_{i+1}, p, r)$ means that p starts a computation to access r in $\gamma_i \mapsto \gamma_{i+1}$.
- $Result(\gamma_i \dots \gamma_j, p, r)$ means that p obtains access to r in $\gamma_{i-1} \mapsto \gamma_i$ and p ends the computation in $\gamma_j \mapsto \gamma_{j+1}$. Notably, p released r between γ_i and γ_j.
- $NoConflict(\gamma_i, p)$ means that, in γ_i, if a resource is allocated to p, then none of its neighbors is using a conflicting resource.

These predicates will be instantiated with the variables of the local resource allocation algorithm. Below, we define the guaranteed service specification of LRA.

Specification 1 (Local Resource Allocation)**.** Let $\mathcal{A}$ be an algorithm. An execution $e = (\gamma_i)_{i\geq 0}$ of $\mathcal{A}$ satisfies the guaranteed service specification of LRA, noted SP_{LRA}, if the three following properties hold:

Resource Conflict Freedom: If a process p starts a computation to access a resource, then there is no conflict involving p during the computation:

$$\forall k \geq 0, \forall k' > k, \forall p \in V, \forall r \in \mathcal{R}_p, \big[Result(\gamma_k \dots \gamma_{k'}, p, r) \wedge \big(\exists l < k, Start(\gamma_l, \gamma_{l+1}, p, r)\big)\big] \\ \Rightarrow \big[\forall i \in \{k, \dots, k'\}, NoConflict(\gamma_i, p)\big]$$

Computation Start: If an application at process p requests resource r, then p eventually starts a computation to obtain r:

$$\forall k \geq 0, \forall p \in V, \forall r \in \mathcal{R}_p, \left[\exists l > k, Request(\gamma_l, p, r) \Rightarrow Start(\gamma_l, \gamma_{l+1}, p, r)\right]$$

Computation End: If process p starts a computation to obtain resource r, the computation eventually ends (in particular, p obtained r during the computation):

$$\forall k \geq 0, \forall p \in V, \forall r \in \mathcal{R}_p, Start(\gamma_k, \gamma_{k+1}, p, r) \Rightarrow \left[\exists l > k, \exists l' > l, Result(\gamma_l \ldots \gamma_{l'}, p, r)\right]$$

Thus, an algorithm $\mathcal{A}$ is snap-stabilizing *w.r.t.* SP_{LRA} (*i.e.*, snap-stabilizing for LRA) if starting from any arbitrary configuration, all its executions satisfy SP_{LRA}.[2]

3 Maximal-Concurrency

Many existing resource allocation algorithms, especially self-stabilizing ones [4, 6, 8, 20, 22, 24, 26], do not consider the concurrency issue. In [19], authors propose a concurrency property *ad hoc* to ℓ-exclusion. We now define the *maximal-concurrency*, which generalizes the definition of [19] to any resource allocation problem.

3.1 Definition

Informally, maximal-concurrency can be defined as follows: if there are processes that can access some resource they are requesting without violating the safety of the considered resource allocation problem, then at least one of them should eventually access one of its requested resources, even if no process releases the resource it holds in the meantime.

Let $P_{CS}(\gamma)$ be the set of processes that are executing their critical section in γ, *i.e.*, the set of processes holding resources in γ. Let $P_{Req}(\gamma)$ be the set of requesting processes that are not in critical section in γ. Let $P_{Free}(\gamma) \subseteq P_{Req}(\gamma)$ be the set of requesting processes that can access their requested resource(s) in γ without violating the safety of the considered resource allocation problem. Let

$$\begin{aligned} continuousCS(\gamma_i \ldots \gamma_j) &\equiv \forall k \in \{i+1, \ldots, j\}, P_{CS}(\gamma_{k-1}) \subseteq P_{CS}(\gamma_k) \\ noReq(\gamma_i \ldots \gamma_j) &\equiv \forall k \in \{i+1, \ldots, j\}, P_{Req}(\gamma_k) \subseteq P_{Req}(\gamma_{k-1}) \end{aligned}$$

The first (resp. second) predicate means that no resource is released (resp. no request occurs)

[2] By contrast, a non-stabilizing algorithm achieves LRA if all its executions starting from *predefined initial* configurations satisfy SP_{LRA}.

between γ_i and γ_j. Notice that for any $i \geq 0$, $continuousCS(\gamma_i \dots \gamma_i)$ and $noReq(\gamma_i \dots \gamma_i)$ trivially hold.

Let $e = (\gamma_i)_{i\geq 0}$, $k \geq 0$ and $T \geq 0$. The function $R(e, k, T)$ is defined when the execution $(\gamma_i)_{i\geq k}$ contains at least T rounds. It is undefined otherwise. In the case it is defined, it returns $x \geq i$ such that the piece of execution $(\gamma_i, ..., \gamma_x)$ contains exactly T rounds.

Definition 1 (Maximal-Concurrency)**.** A resource allocation algorithm is *maximal-concurrent* if and only if for any network,

No Deadlock For every configuration γ such that $pFree(\gamma) \neq \emptyset$, there exists a configuration γ' and a step $\gamma \mapsto \gamma'$ such that $continuousCS(\gamma \dots \gamma') \wedge noReq(\gamma \dots \gamma')$;

No Livelock There exists a number of rounds $N > 0$ such that for every execution $e = (\gamma_i)_{i\geq 0} \in \mathcal{E}$ and for every index $i \geq 0$, if $R(e, i, N)$ exists, then

$$\begin{array}{c} \big(noReq(\gamma_i \dots \gamma_{R(e,i,N)}) \wedge continuousCS(\gamma_i \dots \gamma_{R(e,i,N)}) \wedge P_{Free}(\gamma_i) \neq \emptyset\big) \\ \Rightarrow \qquad (\exists k \in \{i, \dots, R(e, i, N) - 1\}, \exists p \in V, p \in P_{Free}(\gamma_k) \cap P_{CS}(\gamma_{k+1})) \end{array}$$

The property is split into two. **No Deadlock** ensures that whenever requesting processes can be served, the algorithm has no deadlock and can still execute some step (even if the application does not release some resource or request some other). **No Livelock** assumes a given number of rounds, N (which depends on the complexity of the algorithm, and henceforth on the network dimensions): if during an execution, there exists some requesting processes, then at most N rounds later, the algorithm should have served one, even if the application does not release any resource in the meantime. We also assume that no request occurs meanwhile: this allows N to uniquely depend on the algorithm and network; if not, N should also depend on when requests may occur. **KA:** TODO: derniere explication a mieux faire !

3.2 Examples of Instantiation

The examples below show the versatility of our property: we instantiate the set P_{Free} according to the considered problem. Note that the first problem is local, whereas others are not.

Example 1: Local Resource Allocation Maximal-Concurrency. In the local resource allocation problem, a requesting process is allowed to enter its critical section if all its neighbors in critical section are using resources which are compatible with its request. Below, we denote by $\gamma(p).req$ the resource(s) requested by process p in γ. Hence,

$$P_{Free}(\gamma) = \{p \in P_{Req}(\gamma) \mid \forall q \in \mathcal{N}_p, (q \in P_{CS}(\gamma) \Rightarrow \gamma(q).req \rightleftharpoons \gamma(p).req)\}$$

Example 2: ℓ-Exclusion Maximal-Concurrency. The ℓ-exclusion problem [19] is a generalization of mutual exclusion, where up to $\ell \geq 1$ critical sections can be executed concurrently. Solving this problem allows management of a pool of ℓ identical units of a non-sharable reusable resource. Hence,

$$P_{Free}(\gamma) = \emptyset \text{ if } |P_{CS}(\gamma)| = \ell; \quad P_{Free}(\gamma) = P_{Req}(\gamma) \text{ otherwise}$$

Using this latter instantiation, we obtain a definition of maximal concurrency which is equivalent to the "avoiding ℓ-deadlock" property of Fischer *et al.* [19].

Example 3: k-out-of-ℓ Exclusion Maximal-Concurrency. The k-out-of-ℓ exclusion problem [11] is a generalization of the ℓ-exclusion problem where each process can hold up to $k \leq \ell$ identical units of a non-sharable reusable resource. In this context, rather than being the resource(s) requested by process p, $\gamma(p).req$ is assumed to be the number of requested units. Let $Available(\gamma) = \ell - \sum_{p \in P_{CS}(\gamma)} \gamma(p).req$ be the number of available units. Hence,

$$P_{Free}(\gamma) = \{p \in P_{Req}(\gamma) : \gamma(p).req \leq Available(\gamma)\}$$

Using this latter instantiation, we obtain a definition of maximal concurrency which is equivalent to the "strict (k, ℓ)-liveness" property of Datta *et al.* [11], which basically means that if *at least one* request can be satisfied using the available resources, then eventually one of them is satisfied, even if no process releases resources in the mean time.

In the same paper, the authors show the impossibility of designing a k-out-of-ℓ exclusion algorithm satisfying the strict (k, ℓ)-liveness. To circumvent this impossibility, they then propose a weaker property called "(k, ℓ)-liveness", which means that if *any* request can be satisfied using the available resources, then eventually one of them is satisfied, even if no process releases resources in the mean time. Despite this property is weaker than maximal-concurrency, it can be expressed using our formalism as follows:

$$P_{Free}(\gamma) = \emptyset \text{ if } \exists p \in P_{Req}(\gamma), \gamma(p).req > Available(\gamma); \quad P_{Free}(\gamma) = P_{Req}(\gamma) \text{ otherwise}$$

This might seem surprising, but observe that in the above formula, the set P_{Free} is distorted from its original meaning.

3.3 Alternative Definition of Maximal-Concurrency

We now provide an alternative definition of maximal-concurrency: instead of constraining P_{Free} to decrease by one process each N steps, it expresses that the algorithm empties P_{Free}

after enough steps.

We introduce first some notations: let $e = (\gamma_i)_{i\geq 0}$ be an execution and $i \geq 0$ be the index of configuration γ_i. We note $endCS(e,i)$ (resp. $M(e,i)$) the last configuration index such that no resource is released (resp. no request occurs and no resource is released) during the piece of execution $\gamma_i \dots \gamma_{endCS(e,i)}$ (resp. $\gamma_i \dots \gamma_{M(e,i)}$). Formally,

$$\begin{aligned} endCS(e,i) &= \max\{j \geq i : continuousCS(\gamma_i \dots \gamma_j)\} \\ M(e,i) &= \max\{j \geq i : noReq(\gamma_i \dots \gamma_j) \wedge j \leq endCS(e,i)\} \end{aligned}$$

Note that $endCS(e,i)$ is always defined (for any e and any i) since $continuousCS(\gamma_i \dots \gamma_i)$ holds and since we assumed that any critical section lasts finite time. As a consequence, $M(e,i)$ exists, since the set $\{j \geq i : noReq(\gamma_i \dots \gamma_j) \wedge j \leq endCS(e,i)\}$ is not empty ($continuousCS(\gamma_i \dots \gamma_i)$ holds) and is bounded by $endCS(e,i)$.

The maximal-concurrency property can also be defined using the following alternative definition:

Definition 2 (Maximal-Concurrency)**.** A resource allocation algorithm is *maximal-concurrent* if and only if for any network,

No Deadlock For every configuration γ such that $P_{Free}(\gamma) \neq \emptyset$, there exists a configuration γ' and a step $\gamma \mapsto \gamma'$ such that $continuousCS(\gamma \dots \gamma') \wedge noReq(\gamma \dots \gamma')$;

No Livelock There exists a number of rounds $T_{MC} > 0$ such that for every execution $e = (\gamma_i)_{i\geq 0} \in \mathcal{E}$ and for every index $i \geq 0$, if $R(e,i,T_{MC})$ exists, then

$$R(e,i,T_{MC}) \leq M(e,i) \Rightarrow P_{Free}(\gamma_{R(e,i,T_{MC})}) = \emptyset$$

Again, the property is split into two: **No Deadlock** is the same as in Definition 1 but **No Livelock** assumes now a (greater) number of rounds, T_{MC}, such that if the application does no release nor request during T_{MC} rounds, the set P_{Free} becomes empty. As in the former definition, T_{MC} depends on the complexity of the algorithm. Definition 2 is illustrated by Figure 1.

Remark 1. As a direct consequence of **No Livelock**, for every execution $e = (\gamma_i)_{i\geq 0} \in \mathcal{E}$ and for every index $i \geq 0$ such that $R(e,i,T_{MC})$, we have $\forall j \in \{R(e,i,T_{MC}), ..., M(e,i)\}, P_{Free}(\gamma_j) = \emptyset$, since P_{Free} cannot increase while no request is emitted.

Lemma 1. *Definition 1 and Definition 2 are equivalent.*

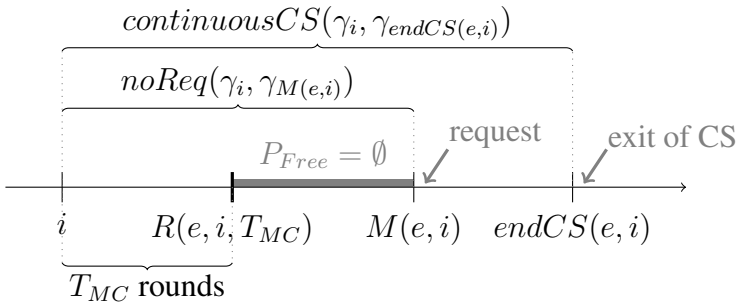


Figure 1: Illustration of Definition 2

Proof. Note first that the **No Deadlock** part is exactly the same in both definitions. Hence we prove the equivalence focusing on the **No Livelock** part.

Assume Definition 1 holds. Let $N > 0$ such that Definition 1 holds. Let $e = (\gamma_i)_{i\geq 0} \in \mathcal{E}$, $i \geq 0$. We pose $T_{MC} = n \times N$. If $R(e,i,T_{MC})$ does not exist or $R(e,i,T_{MC}) > M(e,i)$, then the property holds by vacuous implication. Assume now that $R(e,i,T_{MC})$ exists and $R(e,i,T_{MC}) \leq M(e,i)$. No process can enter P_{Free} between γ_i and $\gamma_{R(e,i,T_{MC})}$ since $R(e,i,T_{MC}) \leq M(e,i)$ and, by definition of $M(e,i)$, there is no new request between γ_i and $\gamma_{M(e,i)}$; hence $noReq(\gamma_i \dots \gamma_{R(e,i,T_{MC})})$. Furthermore, no process leaves its critical section in the meantime because $M(e,i) \leq endCS(e,i)$: $continuousCS(\gamma_i \dots \gamma_{R(e,i,T_{MC})})$. Now, $|P_{Free}(\gamma_i)| \leq n$ and, by Definition 1, at least each N rounds, some process leaves P_{Free} if it is not empty. Hence, as no new process can enter P_{Free}, $P_{Free}(\gamma_{R(e,i,T_{MC})})$ is necessarily empty.

Conversely, assume Definition 2 holds. Let $T_{MC} > 0$ such that Definition 2 holds. Let $e = (\gamma_i)_{i\geq 0} \in \mathcal{E}$, $i \geq 0$. We pose $N = T_{MC}$. If $R(e,i,N)$ does not exists or $\neg noReq(\gamma_i \dots \gamma_{i+N})$, or $\neg continuousCS(\gamma_i \dots \gamma_{i+N})$, or $P_{Free}(\gamma_i) = \emptyset$, then Definition 1 holds by vacuous implication. Now, assume $R(e,i,N)$ exists and $noReq(\gamma_i \dots \gamma_{R(e,i,N)}) \wedge continuousCS(\gamma_i \dots \gamma_{R(e,i,N)}) \wedge P_{Free}(\gamma_i) \neq \emptyset$ holds. This implies that $R(e,i,N) \leq M(e,i)$; hence applying Definition 2, we get $P_{Free}(\gamma_{R(e,i,N)}) = \emptyset$. As $noReq(\gamma_i \dots \gamma_{R(e,i,N)})$ holds, no process has been added to the set P_{Free} in the meantime. Hence, there exists a configuration γ_k with $k \in \{i, ..., R(e,i,N) - 1\}$, such that at least some process $p \in P_{Free}(\gamma_k)$ went out P_{Free}: $p \notin P_{Free}(\gamma_{k+1})$, hence p had access to its resource ($p \in P_{CS}(\gamma_{k+1})$). □

Using the latter definition, remark that an algorithm is not maximal-concurrent if and only if there exists a network for which

- either **No Deadlock** is violated, namely, there exists a configuration γ such that $P_{Free}(\gamma) \neq \emptyset$ and for every configuration γ' such that $continuousCS(\gamma \dots \gamma') \wedge noReq(\gamma \dots \gamma')$, there is no possible step of the algorithm from γ to γ' ($\gamma \not\mapsto \gamma'$).
- or **No Livelock** is violated: $\forall T > 0$, $\exists e = (\gamma_i)_{i\geq 0} \in \mathcal{E}$, $\exists i \geq 0$, $R(e,i,T)$ exists and $R(e,i,T) \leq M(e,i) \wedge P_{Free}(\gamma_{R(e,i,T)}) \neq \emptyset$

4 Maximal-Concurrency versus Fairness

4.1 Necessary Condition on Concurrency in LRA

It is not always possible to ensure the maximal degree of concurrency without violating correctness, more precisely the liveness, of the resource allocation problem. For example, Datta *et al.* showed in [11] that it is impossible to design a k-out-of-ℓ exclusion algorithm that satisfies the strict (k, ℓ)-liveness, equivalent to the maximal concurrency. Hence, we search the maximum degree of concurrency that can be ensured by a LRA algorithm.

Definition 3 below gives a definition of fairness classically used in resource allocation problems. Notably, Computation Start and End properties of Specification 1 trivially implies this fairness property. Next, Lemma 2 states that no LRA algorithm (stabilizing or not) can empty the set P_{Free} in particular existing execution. Actually, this is not possible without violating fairness. This technical result is then used to show that no local resource allocation algorithm can achieve maximal concurrency (Theorem 1).

Definition 3 (Fairness)**.** Each time a process is (continuously) requesting a resource r, it eventually accesses r.

We define the *conflicting neighborhood* of a process p in a configuration γ, denoted $\mathcal{CN}_p(\gamma)$, as the subset of neighbors of p that request conflicting resources in γ, *i.e.*, $\mathcal{CN}_p(\gamma) = \{q \in \mathcal{N}_p : \gamma(p).req \not\rightleftharpoons \gamma(q).req\}$, where $\gamma(p).req$ is the resource requested/used by p in γ. If p is not requesting or in critical section, we assume $\gamma(p).req = \perp$, and $\perp$ is compatible with every resource. Note that if p is not requesting, $\mathcal{CN}_p(\gamma)$ is empty.

Lemma 2. *For any local resource allocation algorithm such that every process can request the same set of resources $\mathcal{R}$ (i.e., $\forall p \in V, \mathcal{R}_p = \mathcal{R}$), for any network, for any process $p \in V$, there exist an execution $e = (\gamma_i)_{i\geq 0} \in \mathcal{E}$, with configuration γ_T, $T \geq 0$, and a node $q \in \mathcal{CN}_p(\gamma_T)$ such that*

$$\mathcal{N}_p \backslash (\{q\} \cup \mathcal{N}_q) = \mathcal{CN}_p(\gamma_T) \backslash \big(\{q\} \cup \mathcal{CN}_q(\gamma_T)\big) = P_{Free}(\gamma_T) \tag{1}$$

and for every execution $e' = (\gamma'_i)_{i\geq 0} \in \mathcal{E}$ which shares the same prefix as e between γ_0 and γ_T (i.e., $\forall i \in \{0, ...T\}, \gamma_i = \gamma'_i$),

$$\forall T' \in \{T, ..., M(e', T)\}, P_{Free}(\gamma_T) = P_{Free}(\gamma'_{T'}) \tag{2}$$

Proof. Consider a local resource allocation algorithm for which every process can request the same set of resources $\mathcal{R}$ and a network. Let $p \in V$. (If p is the unique process of the network, both Equation 1 and 2 holds, trivially.)

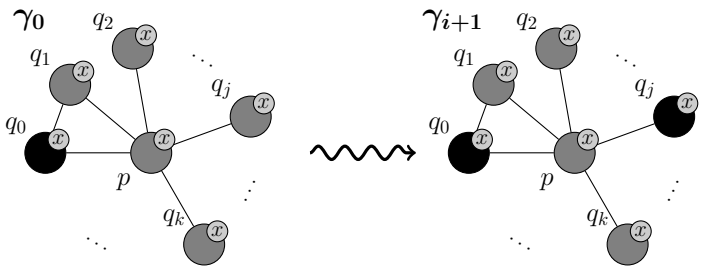


Figure 2: Outline of the execution $(\gamma_i)_{i\geq 0}$ of the proof of Lemma 2 on the neighborhood of p. Black nodes are in critical section, gray nodes are requesting.

First, consider the case when p has a unique neighbor q. Equation 1 trivially holds for any configuration γ_T since $\mathcal{N}_p \backslash (\{q\} \cup \mathcal{N}_q) = \mathcal{CN}_p(\gamma_T) \backslash \big(\{q\} \cup \mathcal{CN}_q(\gamma_T)\big)$ is empty. We pose $T = 0$ and γ_0 a configuration such that p is requesting a resource and q holds a resource conflicting with the resource requested by p and the common neighbors of p and q are requesting resources that are conflicting with the resource holds by q. In γ_0, no other process is either requesting or executing critical section, namely, $P_{Free}(\gamma_0) = \emptyset$ and $P_{Req}(\gamma_0) = \{p\}$. Then, for any possible execution from γ_0, as long as no request occurs and as long as q holds its resource, P_{Free} remains empty: this proves Equation 2.

Second, we assume that p has at least two neighbors. We note $\mathcal{N}_p$ as $\{q_0, ..., q_k\}$ with $k \geq 1$. We fix γ_0 such that

- q_0 holds some resource x such that x is conflicting with x,
- p requests resource x,
- for all $j \in \{1, ..., k\}$, q_j requests resource x,
- no other process is either requesting or executing critical section,

namely, $P_{Free}(\gamma_0) = \mathcal{CN}_p(\gamma_0) \backslash \big((\{q_0\} \cup \mathcal{CN}_{q_0}(\gamma_0)\big) = \mathcal{N}_p \backslash (\{q_0\} \cup \mathcal{N}_{q_0})$ and $P_{Req}(\gamma_0) = \{p\} \cup \{q_1, ..., q_k\}$. See Figure 2-$\gamma_0$.

Again, if $P_{Free}(\gamma_0) = \emptyset$, then we pose $T = 0$, which proves Equation 1. In this case, every q_j, with $j \in \{1, ..., k\}$ is a neighbor of q_0, hence, for any possible execution from γ_0, as long as no request occurs and as long as q_0 holds its resource, P_{Free} remains empty: this proves Equation 2.

Now, we assume that $P_{Free}(\gamma_0) \neq \emptyset$. We build an execution by letting the algorithm execute, while maintaining no request and q_0 in critical section. We do this until some neighbor of p exits P_{Free}: let say this occurs for the first time for q_j at step $\gamma_i \mapsto \gamma_{i+1}$, $i > 0$ and $j \in \{1, ..., k\}$. If this never occurs, we are done since Equation 1 and 2 are both satisfied. Then, assume i and j exist as defined above. We redefine step $\gamma_i \mapsto \gamma_{i+1}$ as follows:

- q_j leaves $P_{Free}(\gamma_i)$ and has access to x (by assumption),

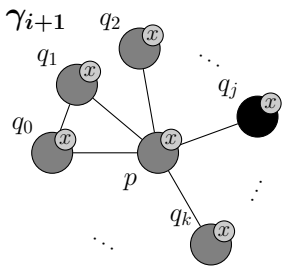


Figure 3: Redefinition of configuration γ_{i+1} in the proof of Lemma 2 on the neighborhood of p.

- q_0 releases its critical section and requests again x.

See Figure 3. Hence, $P_{Req}(\gamma_{i+1}) = \{p\} \cup \{q_l, l \neq j \wedge l \in \{0, ..., k\}\}$ and $P_{Free}(\gamma_{i+1}) = \mathcal{CN}_p(\gamma_{i+1})\backslash\big(\{q_j\} \cup \mathcal{CN}_{q_j}(\gamma_{i+1})\big) = \mathcal{N}_p\backslash(\{q_j\} \cup \mathcal{N}_{q_j})$. The execution reaches configuration γ_{i+1} which is similar to γ_0.

If this scenario is repeated infinitely, the algorithm fails to serve p since it can never access its requested resource. Hence, there exists a configuration γ_T, $T \geq 0$ after which P_{Free} remains equal to $\mathcal{CN}_p(\gamma_T)\backslash\big(\{q_l\} \cup \mathcal{CN}_{q_l}(\gamma_T)\big) = \mathcal{N}_p\backslash(\{q_l\} \cup \mathcal{N}_{q_l})$ (this proves Equation 1) and constant for some $q_l \in \mathcal{N}_p$, until q_l releases its resource or some new request occurs (this proves Equation 2). □

4.2 Impossibility Result

Theorem 1. *It is impossible to design, for any network, a maximal-concurrent local resource allocation algorithm such that every process can request the same set of resources $\mathcal{R}$ (i.e., $\forall p \in V, \mathcal{R}_p = \mathcal{R}$).*

Proof. Consider a network which contains a node $p \in V$, such that $\forall q \in \mathcal{N}_p, \mathcal{N}_p\backslash(\{q\} \cup \mathcal{N}_q) \neq \emptyset$. (It exists, take for instance a star network where p is at the center.)

Now assume a maximal-concurrent local resource allocation algorithm such that every process can request the same set of resources $\mathcal{R}$.

From Lemma 2, there exist $e = (\gamma_i)_{i\geq 0}$ with a configuration γ_T, $T \geq 0$, and $q \in \mathcal{CN}_p(\gamma_T) \subseteq \mathcal{N}_p$ such that $P_{Free}(\gamma_T) = \mathcal{N}_p\backslash(\{q\} \cup \mathcal{N}_q)$. Furthermore, for every execution $e' = (\gamma'_i)_{i\geq 0} \in \mathcal{E}$ which shares the same prefix as e between γ_0 and γ_T, $\forall T' \in \{T, ..., M(e', T)\}, P_{Free}(\gamma_T) = P_{Free}(\gamma'_{T'})$.

Using **No Livelock** from maximal concurrency, there also exists $T_{MC} > 0$ such that for every execution $e' = (\gamma'_i)_{i\geq 0}$, if $R(e', T, T_{MC})$ exists and $R(e', T, T_{MC}) \leq M(e', T)$ then $P_{Free}(\gamma'_{R(e,T,T_{MC})}) = \emptyset$.

We build an execution e' with prefix $\gamma_0...\gamma_T$. From γ_T, we are able to a add step of the algorithm such that no request occurs and no resource is released. This is possible due to

No Deadlock from maximal concurrency and since $P_{Free}(\gamma_T) \neq \emptyset$. Using the second part of Lemma 2, this ensures that $P_{Free}(\gamma'_{T+1}) = P_{Free}(\gamma_T) \neq \emptyset$. We repeat this operation until T_{MC} rounds have elapsed (this is possible since we assumed a weakly fair daemon), so that: $R(e', T, T_{MC}) \leq M(e', T)$. Hence, $P_{Free}(\gamma'_{R(e',T,T_{MC})}) = P_{Free}(\gamma_T) \neq \emptyset$.

Now, using **No Livelock** from maximal concurrency, we also have that $P_{Free}(\gamma'_{R(e',T,T_{MC})}) = \emptyset$, a contradiction. □

5 Partial Concurrency

We generalize the maximal-concurrency to be able to define weaker degree of concurrency. This generalization is called *partial concurrency*.

5.1 Definition

Maximal-concurrency requires that any requesting process eventually accesses its critical section if it can do so without violating safety. Instead, the idea of partial concurrency is to relax slightly this property and to prevent only a small subset of requesting processes to enter their critical section. We define $\mathcal{P}$ as a predicate which represents the sets of requesting processes that may no be served by the algorithm, even if accessing their resource does not violate safety.

Definition 4 (Partial Concurrency *w.r.t.* $\mathcal{P}$)**.** A resource allocation algorithm $\mathcal{A}$ is *partially concurrent* w.r.t. $\mathcal{P}$ if and only if for any network,

No Deadlock For every subset of processes $X \subseteq V$, for every configuration γ, if $\mathcal{P}(X, \gamma)$ holds and $pFree(\gamma) \not\subseteq X$, there exists a configuration γ' and a step $\gamma \mapsto \gamma'$ such that $continuousCS(\gamma \dots \gamma') \wedge noReq(\gamma \dots \gamma')$;

No Livelock There exists a number of rounds $T_{PC} > 0$ such that for every execution $e = (\gamma_i)_{i \geq 0} \in \mathcal{E}$ and for every index $i \geq 0$, if $R(e, i, T_{PC})$ exists then

$$R(e, i, T_{PC}) \leq M(e, i) \Rightarrow \exists X, \mathcal{P}(X, \gamma_{R(e,i,T_{PC})}) \wedge P_{Free}(\gamma_{R(e,i,T_{PC})}) \subseteq X$$

Notice that partial concurrency *w.r.t.* $\mathcal{P}_{max}$ is exactly equivalent to maximal-concurrency where $\forall X \subseteq V, \forall \gamma \in \mathcal{C}, \mathcal{P}_{max}(X, \gamma) \equiv X = \emptyset$.

Remark 2. Again, as a direct consequence of **No Livelock**, for every execution $e = (\gamma_i)_{i \geq 0} \in \mathcal{E}$ and for every index $i \geq 0$ such that $R(e, i, T_{PC})$ exists, we have $\forall j \in \{R(e, i, T_{PC}), ..., M(e, i)\}$, $P_{Free}(\gamma_j) = \emptyset$ since P_{Free} cannot increase while no request is emitted.

5.2 Strong Concurrency

Lemma 2 states that there are always executions where a LRA algorithm, with the same set of resources for each process, must prevent a neighborhood minus one neighbor and its neighborhood to enter critical section in order to ensure fairness. We define the *strong concurrency* as follows:

Definition 5 (Strong Concurrency)**.** A resource allocation algorithm $\mathcal{A}$ is *strongly concurrent* if and only if $\mathcal{A}$ is partially concurrent *w.r.t.* $\mathcal{P}_{strong}$, where $\forall X \subseteq V, \forall \gamma \in \mathcal{C}$,

$$\mathcal{P}_{strong}(X, \gamma) \equiv \exists p \in V, \exists q \in \mathcal{CN}_p(\gamma), X = \mathcal{CN}_p(\gamma) \backslash \big(\{q\} \cup \mathcal{CN}_q(\gamma)\big)$$

A strongly concurrent algorithm prevents at most a neighborhood minus one process and its neighborhood to enter in critical section. Hence, this property is very closed to the maximum degree of concurrency that can be ensured by a LRA algorithm with the same set of resources for each process. In Section 6, we propose a strongly concurrent LRA algorithm.

6 Local Resource Allocation Algorithm

We now propose a snap-stabilizing LRA algorithm which achieves strong concurrency. This algorithm consists of two modules: Algorithm $\mathcal{LRA}$, which manages local resource allocation, and Algorithm $\mathcal{TC}$ which provides a self-stabilizing token circulation service to $\mathcal{LRA}$, whose goal is to ensure fairness.

6.1 Composition

These two modules are composed using a *fair composition* [17], denoted by $\mathcal{LRA} \circ \mathcal{TC}$. In such a composition, each process executes a step of each algorithm alternately.

Notice that the purpose of this composition is only to simplify the design of the algorithm: a composite algorithm written in the locally shared memory model can be translated into an equivalent non-composite algorithm.

Consider the fair composition of two algorithms $\mathcal{A}$ and $\mathcal{B}$. The equivalent non-composite algorithm $\mathcal{C}$ can be obtain by applying the following rewriting rule: In $\mathcal{C}$, a process has its variables in $\mathcal{A}$, those in $\mathcal{B}$, and an additional variable $b \in \{1, 2\}$. Assume now that $\mathcal{A}$ is composed of x actions denoted by

$$lbl_i^A : grd_i^A \rightarrow stmt_i^A, \forall i \in \{1, \ldots, x\}$$

and $\mathcal{B}$ is composed of y actions denoted by

$$lbl_j^B : grd_j^B \rightarrow stmt_j^B, \forall j \in \{1, \ldots, y\}$$

Then, $\mathcal{C}$ is composed of the following $m + n + 2$ actions:

$$\forall i \in \{1, \ldots, x\}, lbl_i^A : (b = 1) \wedge grd_i^A \rightarrow stmt_i^A; b \leftarrow 2$$

$$\forall j \in \{1, \ldots, y\}, lbl_j^B : (b = 2) \wedge grd_j^B \rightarrow stmt_j^B; b \leftarrow 1$$

$$lbl_1 : (b = 1) \wedge \neg \bigwedge_{i=1,\ldots,x} \neg grd_i^A \wedge \bigvee_{i=1,\ldots,y} grd_i^B \rightarrow b \leftarrow 2$$

$$lbl_2 : (b = 2) \wedge \bigwedge_{i=1,\ldots,y} \neg grd_i^B \wedge \bigvee_{i=1,\ldots,x} grd_i^A \rightarrow b \leftarrow 1$$

6.2 Token Circulation Module

We assume that $\mathcal{TC}$ is a self-stabilizing black box which allows $\mathcal{LRA}$ to emulate a self-stabilizing token circulation. $\mathcal{TC}$ provides two outputs to each process p in $\mathcal{LRA}$: the predicate $TokenReady(p)$ and the statement $PassToken(p)$. The predicate $TokenReady(p)$ expresses the fact that the process p holds a token and can release it. Note that this interface of $\mathcal{TC}$ allows some process to hold the token without being allowed to release it yet: this may occur, for example, when, before releasing the token, the process has to wait for the network to clean some faults. The statement $PassToken(p)$ can be used to pass the token from p to one of its neighbor. Of course, it should be executed (by $\mathcal{LRA}$) only if $TokenReady(p)$ holds. Precisely, we assume that $\mathcal{TC}$ satisfies the following properties.

Property 1 (Stabilization)**.** *$\mathcal{TC}$ stabilizes,* i.e.*, reaches and remains in configurations where there is a unique token in the network, independently of any call to $PassToken(p)$ at any process p.*

Property 2. *Once $\mathcal{TC}$ has stabilized, $\forall p \in V$, if $TokenReady(p)$ holds, then $TokenReady(p)$ is continuously true until $PassToken(p)$ is invoked.*

Property 3 (Fairness)**.** *Once $\mathcal{TC}$ has stabilized, if $\forall p \in V$, $PassToken(p)$ is invoked in finite time each time $TokenReady(p)$ holds, then $\forall p \in V$, $TokenReady(p)$ holds infinitely often.*

To design $\mathcal{TC}$, we proceed as follows. There exist several self-stabilizing token circulations for arbitrary rooted networks [9, 12, 23] that contain a particular action, $T : TokenReady(p) \rightarrow PassToken(p)$, to pass the token, and that stabilizes independently of the activations of action T. Now, the networks we consider are not rooted, but identified. So, to obtain a self-stabilizing

token circulation for arbitrary identified networks, we can fairly compose any of them with a self-stabilizing leader election algorithm [3, 18, 13, 1] using the following additional rule: if a process considers itself as leader it executes the token circulation program for a root; otherwise it executes the program for a non-root. Finally, we obtain $\mathcal{TC}$ by removing action T from the resulting algorithm, while keeping $TokenReady(p)$ and $PassToken(p)$ as outputs, for every process p.

Remark 3. Following Property 2 and 3, the algorithm, noted $\mathcal{TC}$ *, made of Algorithm $\mathcal{TC}$ where action $T : TokenReady(p) \rightarrow PassToken(p)$ has been added, is a self-stabilizing token circulation.

The algorithm presented in next section for local resource allocation emulates action T using predicate $TokenReady(p)$ and statement $PassToken(p)$ given as inputs.

6.3 Resource Allocation Module

The code of $\mathcal{LRA}$ is given in Algorithm 1. Priorities and guards ensure that actions of Algorithm 1 are mutually exclusive. We now informally describe Algorithm 1, and explain how Specification 1 is instantiated with its variables.

First, a process p interacts with its application through two variables: $p.req \in \mathcal{R}_p \cup \{\perp\}$ and $p.status \in \{\mathsf{Out}, \mathsf{Wait}, \mathsf{In}, \mathsf{Blocked}\}$. $p.req$ can be read and written by the application, but can only be read by p in $\mathcal{LRA}$. Conversely, $p.status$ can be read and written by p in $\mathcal{LRA}$, but the application can only read it. Variable $p.status$ can take the following values:

- Wait, which means that p requests a resource but does not hold it yet;
- Blocked, which means that p requests a resource, but cannot hold it now;
- In, which means that p holds a resource;
- Out, which means that p is currently not involved into an allocation process.

When $p.req = \perp$, this means that no resource is requested. Conversely, when $p.req \in \mathcal{R}_p$, the value of $p.req$ informs p about the resource requested by the application. We assume two properties on $p.req$. Property 4 ensures that the application (1) does not request for resource r' while a computation to access resource r is running, and (2) does not cancel or modify a request before the request is satisfied. Property 5 ensures that any critical section is finite.

Property 4. *$\forall p \in V$, the updates on $p.req$ (by the application) satisfy the following constraints:*

- *The value of $p.req$ can change from $\perp$ to $r \in \mathcal{R}_p$ if and only if $p.status = \mathsf{Out}$,*
- *The value of $p.req$ can change from $r \in \mathcal{R}_p$ to $\perp$ if and only if $p.status = \mathsf{In}$.*

Algorithm 1 Algorithm $\mathcal{LRA}$ for every process p

Variables

$p.status \in \{\mathsf{Out}, \mathsf{Wait}, \mathsf{Blocked}, \mathsf{In}\}$

$p.token \in \mathbb{B}$

Inputs

$p.req \in \mathcal{R}_p \cup \{\perp\}$: Variable from the application

$TokenReady(p)$: Predicate from $\mathcal{TC}$, indicates that p holds the token

$PassToken(p)$: Statement from $\mathcal{TC}$, passes the token to a neighbor

Macros

$$
\begin{array}{lcl}
Candidates(p) & \equiv & \{q \in \mathcal{N}_p \cup \{p\} : q.status = \mathsf{Wait}\} \\
TokenCand(p) & \equiv & \{q \in Candidates(p) : q.token\} \\
Winner(p) & \equiv & \begin{cases} \max\{q \in TokenCand(p)\} & \textbf{if } TokenCand(p) \neq \emptyset, \\ \max\{q \in Candidates(p)\} & \textbf{otherwise} \end{cases}
\end{array}
$$

Predicates

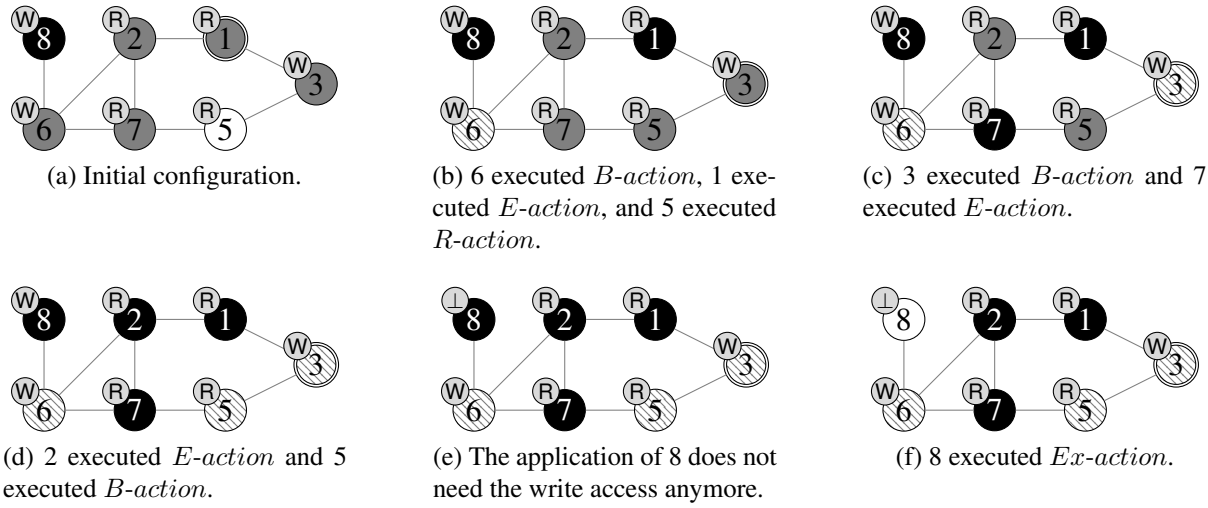
$$
\begin{array}{lcl}
ResourceFree(p) & \equiv & \forall q \in \mathcal{N}_p, \big(q.status = \mathsf{In} \Rightarrow p.req \rightleftharpoons q.req\big) \\
IsBlocked(p) & \equiv & \neg ResourceFree(p) \vee \big(\exists q \in \mathcal{N}_p, q.status = \mathsf{Blocked} \\
 & & \qquad \wedge q.token \wedge p.req \not\rightleftharpoons q.req\big)
\end{array}
$$

Guards

$$
\begin{array}{lcl}
Requested(p) & \equiv & p.status = \mathsf{Out} \wedge p.req \neq \perp \\
Block(p) & \equiv & p.status = \mathsf{Wait} \wedge IsBlocked(p) \\
Unblock(p) & \equiv & p.status = \mathsf{Blocked} \wedge \neg IsBlocked(p) \\
Enter(p) & \equiv & p.status = \mathsf{Wait} \wedge \neg IsBlocked(p) \wedge p = Winner(p) \\
Exit(p) & \equiv & p.status = \mathsf{In} \wedge p.req = \perp \\
ResetToken(p) & \equiv & TokenReady(p) \neq p.token \\
ReleaseToken(p) & \equiv & TokenReady(p) \wedge p.status \in \{\mathsf{Out}, \mathsf{In}\} \wedge \neg Requested(p)
\end{array}
$$

Actions

$$
\begin{array}{llcll}
(1) & RsT\text{-}action & :: & ResetToken(p) & \rightarrow \; p.token \leftarrow TokenReady(p); \\
(2) & Ex\text{-}action & :: & Exit(p) & \rightarrow \; p.status \leftarrow \mathsf{Out}; \\
(3) & RlT\text{-}action & :: & ReleaseToken(p) & \rightarrow \; PassToken(p); \\
(4) & R\text{-}action & :: & Requested(p) & \rightarrow \; p.status \leftarrow \mathsf{Wait}; \\
(4) & B\text{-}action & :: & Block(p) & \rightarrow \; p.status \leftarrow \mathsf{Blocked}; \\
(4) & UB\text{-}action & :: & Unblock(p) & \rightarrow \; p.status \leftarrow \mathsf{Wait}; \\
(4) & E\text{-}action & :: & Enter(p) & \rightarrow \; p.status \leftarrow \mathsf{In}; \\
 & & & & \quad \textbf{if } TokenReady(p) \textbf{ then } PassToken(p);
\end{array}
$$



(a) Initial configuration.

(b) 6 executed $B\text{-}action$, 1 executed $E\text{-}action$, and 5 executed $R\text{-}action$.

(c) 3 executed $B\text{-}action$ and 7 executed $E\text{-}action$.

(d) 2 executed $E\text{-}action$ and 5 executed $B\text{-}action$.

(e) The application of 8 does not need the write access anymore.

(f) 8 executed $Ex\text{-}action$.

Figure 4: Example of execution of $\mathcal{LRA} \circ \mathcal{TC}$.

- *The value of $p.req$ cannot directly change from $r \in \mathcal{R}_p$ to $r' \in \mathcal{R}_p$ with $r' \neq r$.*

Property 5. *$\forall p \in V$, if $p.status = \mathsf{In}$ and $p.req \neq \bot$, then eventually $p.req$ becomes $\bot$.*

Consequently, the predicate $Request(\gamma_i, p, r)$ in Specification 1 is given by $Request(\gamma_i, p, r) \equiv \gamma_i(p).req = r$.

The predicate $NoConflict(\gamma_i, p)$ is expressed by $NoConflict(\gamma_i, p) \equiv \gamma_i(p).status = \mathsf{In} \Rightarrow \big(\forall q \in \mathcal{N}_p, \gamma_i(q).status = \mathsf{In} \Rightarrow (\gamma_i(q).req \rightleftharpoons \gamma_i(p).req)\big)$. (We set $\bot$ compatible with every resource.)

The predicate $Start(\gamma_i, \gamma_{i+1}, p, r)$ becomes true when process p takes the request for resource r into account in $\gamma_i \mapsto \gamma_{i+1}$, *i.e.*, when the status of p switches from Out to Wait in $\gamma_i \mapsto \gamma_{i+1}$ because $p.req = r \neq \bot$ in γ_i: $Start(\gamma_i, \gamma_{i+1}, p, r) \equiv \gamma_i(p).status = \mathsf{Out} \wedge \gamma_{i+1}(p).status = \mathsf{Wait} \wedge \gamma_i(p).req = \gamma_{i+1}(p).req = r$

Assume that $\gamma_i \dots \gamma_j$ is a computation where $Result(\gamma_i \dots \gamma_j, p, r)$ holds: process p accesses resource r, *i.e.*, p switches its status from Wait to In in $\gamma_{i-1} \mapsto \gamma_i$ while $p.req = r$, and later switches its status from In to Out in $\gamma_j \mapsto \gamma_{j+1}$: $Result(\gamma_i \dots \gamma_j, p, r) \equiv \gamma_i(p).status = \mathsf{Wait} \wedge \gamma_{i+1}(p).status = \mathsf{In} \wedge \gamma_i(p).req = \gamma_{i+1}(p).req = r \wedge \gamma_{j-1}(p).status = \mathsf{In} \wedge \gamma_j(p).status = \mathsf{Out}$.

We now illustrate the principles of $\mathcal{LRA}$ with the example given in Figure 4. In this example, we consider the local reader-writer problem. In the figure, the numbers inside the nodes represent their IDs. The color of a node represents its status: white for Out, gray for Wait, black for In, and crossed out for Blocked. A double circled node holds a token. The bubble next to a node represents its request. Recall that we have two resources: R for a reading access and W for a writing access, with R$\rightleftharpoons$R, R$\not\rightleftharpoons$W and W$\not\rightleftharpoons$W.

When the process is idle ($p.status = \mathsf{Out}$), its application can request a resource. In this

case, $p.req \neq \perp$ and p sets $p.status$ to Wait by R-$action$: p starts the computation to obtain the resource. For example, 5 starts a computation to obtain R in (a)↦(b). If one of its neighbors is using a conflicting resource, p cannot satisfy its request yet. So, p switches $p.status$ from Wait to Blocked by B-$action$ (see 6 in (a)↦(b)). If there is no more neighbor using conflicting resources, p gets back to status Wait by UB-$action$.

When several neighbors request for conflicting resources, we break ties using a token-based priority: Each process p has an additional Boolean variable $p.token$ which is used to inform neighbors about whether p holds a token or not. A process p takes priority over any neighbor q if and only if $\big(p.token \wedge \neg q.token\big) \vee \big(p.token = q.token \wedge p > q\big)$. More precisely, if there is no waiting token holder in the neighborhood of p, the highest priority process is the waiting process with highest ID. This highest priority process is $Winner(p)$. Otherwise, the token holders (there may be several tokens during the stabilization phase of $\mathcal{TC}$) blocked all their requesting neighbors, except the ones requesting for non-conflicting resources, and until the token holders obtain their requested resources. This mechanism allows to ensure fairness by slightly decreasing the level of concurrency. (The token circulates to eventually give priority to blocked processes, *e.g.*, processes with small IDs.)

The highest priority waiting process in the neighborhood gets status In and can use its requested resource by E-$action$, *e.g.*, 7 in step (b)↦(c) or 1 in (a)↦(b). Moreover, if it holds a token, it releases it. Notice that, as a process is not blocked when one of its neighbors is using a compatible resource, several neighbors using compatible resources can concurrently enter and/or execute their critical section (see 1, 2, and 7 in Configuration (d)). When the application at process p does not need the resource anymore, *i.e.*, when it sets the value of $p.req$ to $\perp$, p executes Ex-$action$ and switches its status to Out, *e.g.*, 8 during step (e)↦(f).

RlT-$action$ is used to straight away pass the token to a neighbor when the process does not need it, *i.e.*, when either its status is Out and the process does not request any resource or when its status is In. (Hence, the token can eventually reach a requesting process and help it to satisfy its request.)

The last action, RsT-$action$, ensures the consistency of variable $token$ so that the neighbors realize whether or not a process holds a token.

Hence, a requesting process is served in a finite time. This is illustrated by an example of execution on Figure 5. We consider here the local mutual exclusion problem in which two neighbors cannot concurrently execute their critical section. We try to delay as long as possible the service of process 2. As its neighbors 7 and 8 also request the resource but have greater IDs, they can access their critical section before 2 (see Steps (a)↦(b) and (e)↦(f)). But a token circulates in the network and eventually reaches 2 (see Configuration (g)). Then, 2 has priority over its neighbors (even if it has a lower ID) and eventually starts executing its critical section in (j)↦(k).

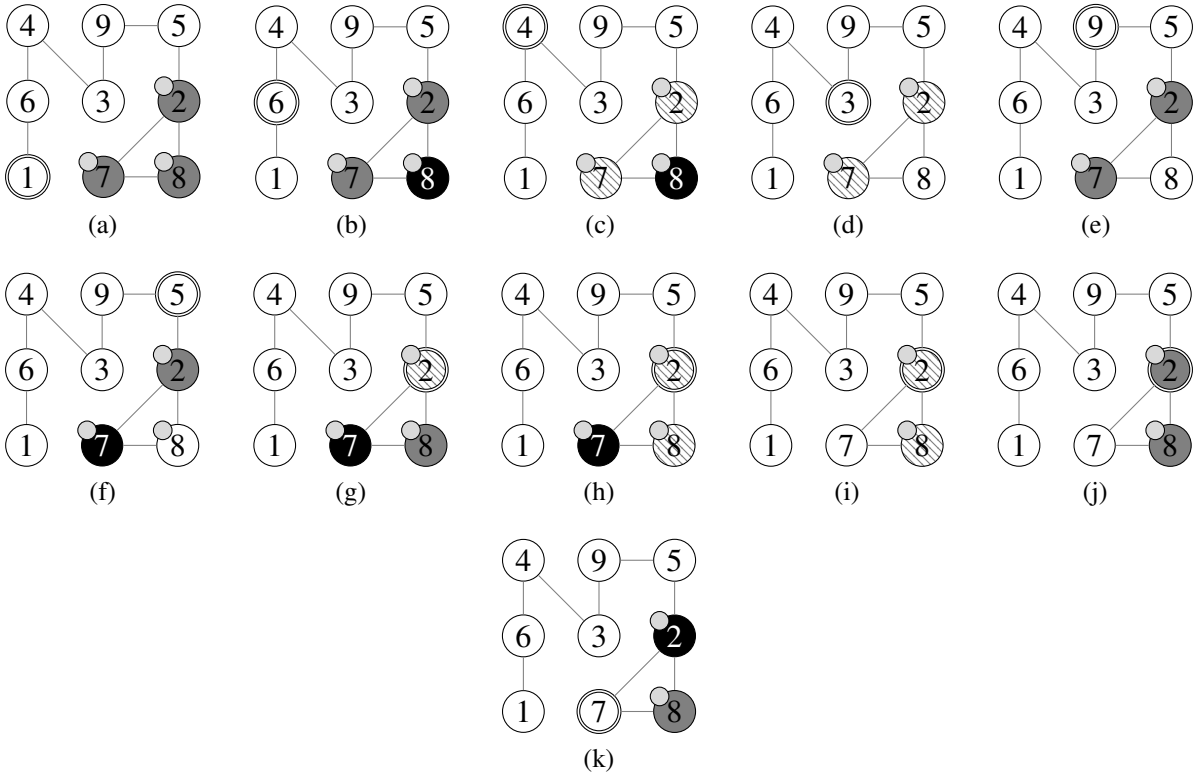


Figure 5: Example of execution of $\mathcal{LRA} \circ \mathcal{TC}$ on the local mutual exclusion problem. The bubbles mark the requesting processes.

7 Correctness and Complexity Analysis of $\mathcal{LRA} \circ \mathcal{TC}$

In this section, we show that $\mathcal{LRA} \circ \mathcal{TC}$ is snap-stabilizing *w.r.t.* SP_{LRA} (Specification 1), assuming a weakly fair daemon.

7.1 Correctness

In this section, we prove that $\mathcal{LRA} \circ \mathcal{TC}$ is snap-stabilizing *w.r.t.* problem of LRA, assuming a distributed weakly fair daemon (see Specification 1), page 8. First, we show the safety part, namely, that the Resource Conflict Freedom property is always satisfied. Then we assume a distributed weakly faire daemon to prove the liveness part, *i.e.*, Computation Start and Computation End properties.

Remark 4. If $E\text{-}action$ is enabled at a process p in a configuration γ, then $\forall q \in \mathcal{N}_p$, $\big(\gamma(q).status = \mathsf{In} \Rightarrow \gamma(p).req \rightleftharpoons \gamma(q).req\big)$.

Lemma 3. *$E\text{-}action$ cannot be simultaneously enabled at two neighbors.*

Proof. Let γ be a configuration. Let $p \in V$ and $q \in \mathcal{N}_p$. Assume by contradiction that $E\text{-}action$ is enabled at p and q in γ. Then, $\gamma(p).status = \gamma(q).status = \mathsf{Wait}$ and both $p = Winner(p)$ and $q = Winner(q)$ hold in γ. Note that by definition, $p, q \in WaitingNeigh(p)$ and $p, q \in WaitingNeigh(q)$. Let examine two cases:

1. If $TokenCand(p) \neq \emptyset$ in γ, as $p = Winner(p)$ in γ, then $p = \max\{x \in TokenCand(p)\}$ and so $\gamma(p).token = \texttt{true}$. Hence, $p \in TokenCand(q) \neq \emptyset$ in γ. So $q = \max\{x \in TokenCand(q)\}$ holds in γ and $\gamma(q).token = \texttt{true}$. (Otherwise, $E\text{-}action$ is not enabled at q in γ.) Now, $p = \max\{x \in TokenCand(p)\} > q$ since $q \in TokenCand(p)$ and $q = \max\{x \in TokenCand(q)\} > p$ since $p \in TokenCand(q)$, a contradiction.

2. If $TokenCand(p) = \emptyset$ in γ, then $\gamma(p).token = \gamma(q).token = \texttt{false}$. Now, as $p = Winner(p)$, $p = \max\{x \in Candidates(p)\} > q$. Hence, $q = \max\{x \in TokenCand(q)\}$, so $\gamma(q).token$ holds, a contradiction.

□

Lemma 4. *Let $\gamma \mapsto \gamma'$ be a step. Let $p \in V$. If $NoConflict(\gamma, p)$ holds, then $NoConflict(\gamma', p)$ holds.*

Proof. Let $\gamma \mapsto \gamma'$ be a step. Let $p \in V$. Assume by contradiction that $NoConflict(\gamma, p)$ holds but $\neg NoConflict(\gamma', p)$. Then, $\gamma'(p).status = \mathsf{In}$ and $\exists q \in \mathcal{N}_p$ such that $\gamma'(q).status = \mathsf{In}$ and $\gamma'(q).req \neq \gamma'(p).req$. As a consequence, $\gamma'(p).req \in \mathcal{R}_p$ and $\gamma'(p).req \in \mathcal{R}_q$.

Using Property 4,

- The value of $p.req$ can change from $\perp$ in γ to $r \in \mathcal{R}_p$ in γ' only in $\gamma(p).status = \mathsf{Out}$. But $\gamma'(p).status = \mathsf{In}$ and there is no way to go from Out to In in one step.
- The value of $p.req$ cannot change from $r' \in \mathcal{R}_p$ in γ to $r \in \mathcal{R}_p$ with $r \neq r'$.

Hence, $\gamma(p).req = \gamma'(p).req \in \mathcal{R}_p$. We can make the same reasoning on q so $\gamma(q).req = \gamma'(q).req \in \mathcal{R}_q$, and $\gamma(q).req \neq\!\!\!\!\rightleftharpoons \gamma(p).req$. Now, there are two cases:

1. If $\gamma(p).status = \mathsf{In}$, as $NoConflict(\gamma, p)$ holds, $\forall x \in \mathcal{N}_p, (\gamma(x).status = \mathsf{In} \Rightarrow \gamma(p).req \rightleftharpoons \gamma(x).req)$. In particular, $\gamma(q).status \neq \mathsf{In}$, since $\gamma(q).req \neq\!\!\!\!\rightleftharpoons \gamma(p).req$. So q executes $E\text{-}action$ $\gamma \mapsto \gamma'$ to obtain $status$ In. This contradicts Remark 4, since q has a conflicting neighbor, p, with status In in γ.
2. If $\gamma(p).status \neq \mathsf{In}$, then p executes $E\text{-}action$ in step $\gamma \mapsto \gamma'$ to get status In. Now, there are two cases:
   (a) If $\gamma(q).status \neq \mathsf{In}$, then q executes $E\text{-}action$ in $\gamma \mapsto \gamma'$. So $E\text{-}action$ is enabled at p and q in γ, a contradiction to Lemma 3.
   (b) If $\gamma(q).status = \mathsf{In}$, then $E\text{-}action$ is enabled at p in γ even though a neighbor of p has $status$ In and a conflicting request (p is in a similar situation to the one of q in case 1), a contradiction to Remark 4.

□

Theorem 2 (Resource Conflict Freedom)**.** *Any execution of $\mathcal{LRA} \circ \mathcal{TC}$ satisfies the resource conflict freedom property.*

Proof. Let $e = (\gamma_i)_{i \geq 0}$ be an execution $\mathcal{LRA} \circ \mathcal{TC}$. Let $k \geq 0$ and $k' > k$. Let $p \in V$. Let $r \in \mathcal{R}_p$. Assume $Result(\gamma_k \dots \gamma_{k'}, p, r)$. Assume $\exists l < k$ such that $Start(\gamma_l, \gamma_{l+1}, p, r)$. In particular, $\gamma_l(p).status \neq \mathsf{In}$. Hence, $NoConflict(\gamma_l, p)$ trivially holds. Using Lemma 4, $\forall i \geq l$, $NoConflict(\gamma_i, p)$ holds. In particular, $\forall i \in \{k, \dots, k'\}$, $NoConflict(\gamma_i, p)$. □

In the following, we assume a weakly fair daemon.

Lemma 5. *The stabilization of $\mathcal{TC}$ is preserved by fair composition.*

Proof. The fair composition of $\mathcal{TC}$ with $\mathcal{LRA}$ does not degrade the behavior of $\mathcal{TC}$ (which is just slowed down). Indeed, its actions are similar (apart from the additional Boolean management) and $\mathcal{TC}$ stabilizes independently to the call to $PassToken$ (Property 1). So $\mathcal{TC}$ remains self-stabilizing in $\mathcal{LRA} \circ \mathcal{TC}$. (But its stabilization phase lasts twice as long.) □

Lemma 6. *A process cannot keep a token forever in $\mathcal{LRA} \circ \mathcal{TC}$.*

Proof. Let e be an infinite execution. By Lemma 5, the token circulation eventually stabilizes, *i.e.*, there is a unique token in every configuration after stabilization of $\mathcal{TC}$. Assume by contradiction that, after such a configuration γ, a process p keeps the token forever: $TokenReady(p)$ holds forever and $\forall q \in V$ with $q \neq p$, $\neg TokenReady(q)$ holds forever.

First, let show that the values of $token$ variables are eventually updated to the corresponding value of the predicate $TokenReady$. Note that the values of predicate $TokenReady$ do not change. So, if there is $x \in V$ such that $x.token \neq TokenReady(x)$, $RsT\text{-}action$ is continuously enabled at x with higher priority. Hence, in finite time, x is selected by the weakly fair daemon and updates its $token$ variable. Therefore, in finite time, the system reaches and remains in configurations where $p.token = \texttt{true}$ forever and $\forall q \in V$ with $q \neq p$, $q.token = \texttt{false}$, forever. Let γ' be such a configuration. Notice that $RsT\text{-}action$ is then continuously disabled from γ'. Then, we can distinguish five cases:

1. If $\gamma'(p).status = \mathsf{Wait}$, $TokenCand(p) = \{p\}$ and so $Winner(p) = p$ holds forever, and $\forall q \in \mathcal{N}_p$, $TokenCand(q) = \{p\}$ and $Winner(q) = p \neq q$ holds forever. So, $\forall q \in \mathcal{N}_p$, $E\text{-}action$ is disabled forever at γ'. Now, if $\exists q \in \mathcal{N}_p$ such that $\gamma'(q).status = \mathsf{In} \wedge \gamma'(q).req \neq \gamma'(p).req$, then, as $\perp$ is compatible with any resource, $\gamma'(q).req \neq \perp$. Using Property 5, in finite time the request of q becomes $\perp$ and remains $\perp$ until q obtains $status$ Out (Property 4). So $Ex\text{-}action$ is continuously enabled at q. Hence, in finite time, those processes leave critical section and cannot enter again since $E\text{-}action$ is disabled forever, and so $\forall q \in \mathcal{N}_p$, $q.status \neq \mathsf{In}$, forever. So $IsBlocked(p)$ does not hold anymore. Notice that, if p gets status Blocked in the meantime, $UB\text{-}action$ is continuously enabled at p, so p gets back status Wait in finite time. Then, $Winner(p) = p$ still holds so $E\text{-}action$ is continuously enabled at p. Hence, in finite time, p executes $E\text{-}action$ and releases its token, a contradiction.

2. If $\gamma'(p).status = \mathsf{Out}$ and $\gamma'(p).req = \perp$, $RlT\text{-}action$ is continuously enabled at p. So, in finite time, p executes $RlT\text{-}action$ and releases its token by a call to $PassToken(p)$, a contradiction.

3. If $\gamma'(p).status = \mathsf{Out}$ and $\gamma'(p).req \neq \perp$, the application cannot change $p.req$ until p enters its critical section (Property 4). Hence, $RlT\text{-}action$ is disabled until p gets status In. So, $R\text{-}action$ is continuously enabled at p and p eventually gets status Wait. We reach case 1 which is not possible.

4. If $\gamma'(p).status = \mathsf{Blocked}$, $\forall q \in \mathcal{N}_p$ such that $\gamma'(p).req \neq \gamma'(q).req$, $IsBlocked(q)$ holds forever so $E\text{-}action$ is disabled at q forever. Now, as in case 1, $\forall q \in \mathcal{N}_p$ such that $\gamma'(p).req \neq \gamma'(q).req$, $q.status \neq \mathsf{In}$ in finite time, and then, $UB\text{-}action$ is continuously enabled at p. In finite time, p gets $status$ Wait and, as in case 1, eventually releases its token, a contradiction.

5. If $\gamma'(p).status = \mathsf{In}$, either $\gamma'(p).req = \perp$ or in finite time $p.req$ becomes $\perp$ (Property 5) and remains $\perp$ until p obtains $status$ Out (Property 4). Then, $Ex\text{-}action$ is continuously enabled at p. So p eventually gets status Out. We reach case 2 and case 3 which are not possible.

□

Lemma 6 implies that the hypothesis of Property 3 is satisfied. Hence, we can deduce Corollary 1.

Corollary 1. *After stabilization of the token circulation module, $TokenReady(p)$ holds infinitely often at any process p in $\mathcal{LRA} \circ \mathcal{TC}$.*

Lemma 7. *A process of $status$* Wait *or* Blocked *executes $E\text{-}action$ in finite time.*

Proof. Let e be an infinite execution, $\gamma \in e$ be a configuration, $p \in V$ such that $\gamma(p).status \in \{\mathsf{Wait}, \mathsf{Blocked}\}$. By Lemma 5, the token circulation eventually stabilizes. By Corollary 1, in finite time p holds the unique token. If p did not execute $E\text{-}action$ yet at this configuration then, it cannot keep forever the token (Lemma 6) and it can only release its token executing $E\text{-}action$ (by Property 2). □

Lemma 8. *A process of $status$* Wait *or* Blocked *gets $status$* Out *in finite time.*

Proof. Let $p \in V$. Let γ be a configuration. If $\gamma(p).status \in \{\mathsf{Wait}, \mathsf{Blocked}\}$, then p executes $E\text{-}action$ in a finite time (Lemma 7) and gets $status$ In in configuration γ'. Now, if $\gamma'(p).req \neq \perp$, in finite time it changes to $\perp$ (Property 5) and cannot change anymore until p gets $status$ Out (Property 4). So, $Exit(p)$ continuously holds and $Ex\text{-}action$ is continuously enabled at p. So, p executes $Ex\text{-}action$ and eventually gets $status$ Out. □

Notice that if a process that had $status$ Wait or Blocked obtains $status$ Out, this means that its computation ended.

Theorem 3 (Computation Start)**.** *Any execution of $\mathcal{LRA} \circ \mathcal{TC}$ satisfies the Computation Start property.*

Proof. Let $e = (\gamma_i)_{i \geq 0}$ be an execution. Let $k \geq 0$. Let $p \in V$. Let $r \in \mathcal{R}_p$. First, p eventually has $status$ Out. Indeed, if $\gamma_k(p).status \neq \mathsf{Out}$, p is computing a service and cannot answer a new request (the application cannot change its request before being served by Property 4). But, in finite time, p ends its current computation and obtains $status$ Out (Lemma 8), let say in $\gamma_{j-1} \mapsto \gamma_j$ ($j \geq k$).

Now, if $\gamma_j(p).req \neq \perp$ holds, it holds continuously (Property 4). p eventually holds the unique token (Corollary 1). Then, $R\text{-}action$ is continuously enabled at p, and, p eventually

executes $R\text{-}action$ (let say in $\gamma_l \mapsto \gamma_{l+1}$, $l \geq j \geq k$). So $\gamma_{l+1}(p).status = \mathsf{Wait}$. Notice that the application of p cannot change its request (Property 4), so $\gamma_l(p).req = \gamma_{l+1}.req = r$. Hence, $Request(\gamma_l, p, r)$ and $Start(\gamma_l, \gamma_{l+1}, p, r)$ hold. □

Theorem 4 (Computation End)**.** *Any execution of $\mathcal{LRA} \circ \mathcal{TC}$ satisfies the Computation End property.*

Proof. Let $e = (\gamma_i)_{i \geq 0}$ be an execution. Let $k \geq 0$. Let $p \in V$. Let $r \in \mathcal{R}_p$. If $Start(\gamma_k, \gamma_{k+1}, p, r)$ holds, then $\gamma_{k+1}(p).status = \mathsf{Wait}$ and $\gamma_{k+1}(p).req = r$. Using Lemma 7, in finite time, p executes $E\text{-}action$ and gets $status$ In (let say in $\gamma_{l-1} \mapsto \gamma_l$, $l > k$). Notice that the application cannot change the value of req until p obtains $status$ In (Property 4) so $\gamma_{l-1}(p).req = \gamma_l(p).req = \gamma_{k+1}(p).req = r$.

Then, in finite time, the application does not need the resource anymore and changes the value of $p.req$ to $\perp$ (Property 5). So $p.req = \perp$ continuously and $Ex\text{-}action$ is continuously enabled at p. Let say p executes $Ex\text{-}action$ in $\gamma_{l'} \mapsto \gamma_{l'+1}$ ($l' \geq l$). So, $\gamma_{l'}(p).status = \mathsf{In}$ and $\gamma_{l'+1}(p).status = \mathsf{Out}$, and consequently, $Result(\gamma_l \ldots \gamma_{l'}, p, r)$ holds. □

Using Theorems 2, 3, and 4, we can conclude:

Theorem 5 (Correctness)**.** *Algorithm $\mathcal{LRA} \circ \mathcal{TC}$ is snap-stabilizing* w.r.t. SP_{LRA} *assuming a distributed weakly fair daemon.*

7.2 Complexity Analysis

In the following subsection, we analyze the waiting time, *i.e.*, the number of rounds needed to obtain critical section after a request. We assume that the execution of critical section *lasts at most one round*.

Lemma 9. *In $\mathcal{LRA} \circ \mathcal{TC}$, after stabilization of $\mathcal{TC}$, there is at most 2×6 rounds between the time when $TokenReady(p)$ becomes true at a process p and the execution of $PassToken(p)$.*

Proof. As $PassToken()$ is only executed on the $\mathcal{LRA}$ part of $\mathcal{LRA} \circ \mathcal{TC}$, we focus on counting rounds from $\mathcal{LRA}$, first. Then the result will be mutliply by 2 due to composition with $\mathcal{TC}$.

Let p be a process. After stabilization of the token circulation algorithm, a process can only release its token executing either $RlT\text{-}action$ or $E\text{-}action$ (Property 2).

Assume $TokenReady(p)$ holds. In one round, the variable $p.token$ is updated from $TokenReady(p)$ executing $RsT\text{-}action$. Then, there are three cases:

1. If the process is requesting but does not get the critical section yet, the worst case is $p.status = \mathsf{Out}$ and $p.req \neq \perp$. In one round, p executes $R\text{-}action$ and gets $status$

Wait. Then, if there are some neighbors of p in critical section that are using a conflicting resource, p executes $B\text{-}action$ in one additional round. In one round, the processes in critical section ends their computation. Notice that, as p holds the unique token, no other neighbor of p can enter the critical section meanwhile. Finally, p is no more blocked and executes $UB\text{-}action$ in one round before executing $E\text{-}action$ in one additional round. Executing $E\text{-}action$, p releases its token after six rounds in total.

2. If $p.status = \mathsf{Out}$ and $p.req = \perp$, the process does not need the token and releases it executing $RlT\text{-}action$ in at most one round.

3. If $p.status = \mathsf{In}$, in one round p executes $Ex\text{-}action$ and get $p.status = \mathsf{Out}$. Then, as in case 2, p releases the token after one additional round that is to say three rounds in total.

□

Let T_S be the stabilization time in rounds of $\mathcal{TC}$. Let T_{tok} be a bound on the number of rounds required to obtain the unique token in $\mathcal{TC}$ * (the algorithm obtained when adding action $T : TokenReady(p) \to PassToken(p)$ to $\mathcal{TC}$, see Remark 3, page 20) after stabilization. Let N_{tok} be a bound on the number of $PassToken$ realized between two consecutive executions of $PassToken$ by the same process.

Theorem 6 (Waiting Time)**.** *Assuming the execution of the critical section lasts at most one round, a requesting process obtains access to critical section in at most* $2(T_s{+}T_{tok}{+}6(N_{tok}{+}1))$ *rounds.*

Proof. Let $p \in V$ such that $p.req \neq \perp$ and $p.status = \mathsf{Out}$. In the worst case, p must wait to hold a token and to be the unique token holder to get its critical section. $\mathcal{TC}$ stabilizes in $2T_S$ rounds (factor 2 comes from composition). Then, in at most $2(T_{tok} + N_{tok} \times 6)$, p gets the token, since it has to wait T_{tok} rounds due to Algorithm $\mathcal{TC}$ and $N_{tok} \times 6$ rounds due to Algorithm $\mathcal{LRA}$ (again, factor 2 comes from composition). Indeed, while executing action $T : TokenReady \to PassToken$ is atomic in $\mathcal{TC}$ *, a process keeps the token at most 2×6 rounds in $\mathcal{LRA} \circ \mathcal{TC}$ (Lemma 9). Finally, to obtain critical section, it is required that p executes $E\text{-}action$ which also releases the token: by Lemma 9, again, this may require 2×6 additional rounds. Hence, in at most $2(T_s + T_{tok} + 6(N_{tok} + 1))$ rounds, p obtains its critical section.

□

For example, if we choose the token circulation algorithm introduced by Cournier *et al.* in [9], T_s is $O(n)$ rounds, T_{tok} is also $O(n)$ rounds, and N_{tok} is $O(n)$ executions of $PassToken$. Applying these results to the previous Theorem 6 shows that the waiting time is $O(n)$ rounds.

8 Strong Concurrency of $\mathcal{LRA} \circ \mathcal{TC}$

We now show that $\mathcal{LRA} \circ \mathcal{TC}$ is strongly concurrent.

We first prove **No Deadlock**.

Lemma 10. *Algorithm $\mathcal{LRA} \circ \mathcal{TC}$ meets the **No Deadlock** property of strong concurrency: for every subset of processes $X \subseteq V$, for every configuration γ, if $\mathcal{P}_{strong}(X, \gamma)$ holds and $pFree(\gamma) \not\subseteq X$, there exists a configuration γ' and a step $\gamma \mapsto \gamma'$ such that $continuousCS(\gamma \dots \gamma') \wedge noReq(\gamma \dots \gamma')$;*

Proof. Assume a configuration γ such that the algorithm can perform no step. Necessarily, this configuration takes place after stabilization of $\mathcal{TC}$ (see Property 1); hence a unique token exists in γ. Notice first that the unique token holder t cannot have status Wait. Otherwise, $t = Winner(\gamma(t))$ and, either $\neg IsBlocked(\gamma(t))$ holds and $E\text{-}action$ is enabled at t, or $B\text{-}action$ is enabled at t, a contradiction.

We show that it exists a process p and one of its neighbor $q \in \mathcal{CN}_q(\gamma)$ such that $P_{Free}(\gamma) \subseteq \mathcal{CN}_p(\gamma) \backslash \big(\{q\} \cup \mathcal{CN}_q(\gamma)\big)$. Indeed, if $P_{Free}(\gamma)$ is empty, we are done. Otherwise, let $r \in P_{Free}(\gamma)$: necessarily, $\gamma(r).status \in \{\mathsf{Wait}, \mathsf{Blocked}\}$ and $\gamma(r).req \neq \bot$.

If $\gamma(r).status = \mathsf{Wait}$, then, $\neg IsBlocked(\gamma(r)) \wedge r \neq Winner(\gamma(r))$ holds. We can build a sequence of processes $r_0, r_1, \dots, r_k$ where $r_0 = r$ and such that $\forall i \in \{1, \dots, k\}$, $r_i = Winner(r_{i-1})$. (Notice that none of the r_i are the token holder, since the token holder does not have status Wait.) This sequence is finite because $r_0 < r_1 < \dots < r_k$ (so a process cannot be involved several times in this sequence) and the number of processes is finite. Hence, we can take this sequence maximal, in which case, $r_k = Winner(r_k)$ and r_k is then enabled, a contradiction.

Hence, $\gamma(r).status = \mathsf{Blocked}$ but $IsBlocked(\gamma(r))$ holds. Note that $ResourceFree(\gamma(r))$ holds since $r \in P_{Free}(\gamma)$. Using $IsBlocked(\gamma(r))$, we have, $\exists p \in \mathcal{N}_r, \gamma(p).status = \mathsf{Blocked} \wedge \gamma(p).token \wedge \gamma(r).req \neq \gamma(p).req$: $r \in \mathcal{CN}_p(\gamma)$ and p is the unique token holder in γ. But p has also status Blocked. As p is not enabled, in this case, $\neg ResourceFree(\gamma(p))$ holds, namely, there exists a process $q \in \mathcal{N}_p$ such that $\gamma(q).status$ is In and $\gamma(p).req \neq \gamma(q).req$; hence q is not in $P_{Free}(\gamma)$ but is in $\mathcal{CN}_p(\gamma)$. Note that r is neither in $\mathcal{CN}_q(\gamma)$ otherwise it could not be in $P_{Free}(\gamma)$. Hence, $r \in \mathcal{CN}_p(\gamma) \backslash \big(\{q\} \cup \mathcal{CN}_q(\gamma)\big)$ and $q \in \mathcal{CN}_p(\gamma)$. □

Second, we prove **No Livelock** in several steps. We first examine the neighbors of the token holder p, after stabilization of $\mathcal{TC}$, and provided that the token remains at p a long time enough.

Lemma 11. *Let $e = (\gamma_j)_{j \geq 0} \in \mathcal{E}$ be an execution of $\mathcal{LRA} \circ \mathcal{TC}$ and let $i \geq 0$ such that $\mathcal{TC}$ has stabilized at γ_i. We note $p \in V$ the unique token holder at γ_i, i.e., $TokenReady(\gamma_i(p))$ holds. If $R(e, i, 6)$ exists and $R(e, i, 6) \leq M(e, i)$, if $\forall j \in \{i, ..., R(e, i, 6)\}$, $PassToken(p)$ is not executed at step $\gamma_j \mapsto \gamma_{j+1}$, then for every $k \in \{R(e, i, 6), ..., M(e, i)\}$,*

- $\gamma_k(p).req \neq \perp$ *and* $\gamma_k(p).status = \mathsf{Blocked}$,
- $\exists q \in \mathcal{CN}_p()$, $\gamma_k(q).req \neq \perp$, $\gamma_k(q).req \neq \gamma_k(p).req$ *and* $\gamma_k(q).status = \mathsf{In}$,
- $\forall r \in P_{Free}(\gamma_k) \cap \mathcal{CN}_p(\gamma_k)$, $r \notin \{q\} \cup \mathcal{CN}_q(\gamma_k)$

Proof. First, notice that last item is direct from the definition of P_{Free}: no process in P_{Free} can be neighbor of a requesting process with status In (hence in critical section) using a conflicting resource.

Let $e = (\gamma_j)_{j\geq 0} \in \mathcal{E}$ be an execution of $\mathcal{LRA} \circ \mathcal{TC}$ and let $i \geq 0$ such that $\mathcal{TC}$ has stabilized at γ_i. Assume $p \in V$ the unique token holder at γ_i. Assume also that $R(e,i,6)$ exists, $R(e,i,6) \leq M(e,i)$, and $\forall j \in \{i, ..., R(e,i,6)\}$, $PassToken(p)$ is not executed at step $\gamma_j \mapsto \gamma_{j+1}$.

If $RsT\text{-}action$ is enabled at $\gamma_i(p)$, then it is continuously enabled and after at most two rounds (because of the composition), the weakly fair daemon ensures that it is executed. Afterwhile, our assumptions ensure that it is disabled. Hence, $TokenReady(\gamma_{R(e,i,2)}(p)) = \gamma_{R(e,i,2)}(p).token = true$.

Now, $Ex\text{-}action$ is disabled $\gamma_{R(e,i,2)}(p)$. Indeed, if enabled, p would get status Out with $p.req = \perp$ at most 2 rounds later; this would activate $RlT\text{-}action$ which, at most 2 more rounds later, would execute $PassToken(p)$. Hence, if $Ex\text{-}action$ is enabled at $\gamma_{R(e,i,2)}(p)$ then $PassToken(p)$ would have been executed at most at $\gamma_{R(e,i,6)}$, a contradiction.

For similar, but simpler reason, $RlT\text{-}action$ and $E\text{-}action$ are disabled at $\gamma_{R(e,i,2)}(p)$. Hence, $\gamma_{R(e,i,2)}(p).status$ cannot be In.

If $\gamma_{R(e,i,2)}(p).status$ is Out, then $\gamma_{R(e,i,2)}(p).req \neq \perp$ (otherwise $RlT\text{-}action$ would be enabled) and $R\text{-}action$ is enabled and, in at most 2 rounds, executed. Afterwhile, $E\text{-}action$ should still be disabled: hence $IsBlocked(p)$ is true (note that $p = Winner(p)$ since p is the unique token holder) and consequently, there exists $q \in \mathcal{CN}_p(\gamma)$ such that $q.status = \mathsf{In}$. This proves that $B\text{-}action$ is enabled and then executed at most two rounds later. At last, focusing on configuration $\gamma_{R(e,i,6)}$, p has necessarily status Blocked and its request is not $\perp$. Furthermore, its conflicting neighbor q has still status In (with conflicting requesting resource), since no release can occur.

If $\gamma_{R(e,i,2)}(p).status$ is Wait, then using the same piece of reasoning, at $\gamma_{R(e,i,4)}$, p has necessarily status Blocked and a conflicting neighbor q with status In. Then $IsBlocked(p)$ remains true, due to q, during the next two rounds: hence at $\gamma_{R(e,i,6)}$, p has status Blocked and a conflicting neighbor q with status In

If $\gamma_{R(e,i,2)}(p).status$ is Blocked, then $IsBlocked(p)$ holds due to some conflicting neighbor with status In. The situation remains so until $\gamma_{R(e,i,6)}$, since no release occurs.

Hence, the three items of Lemma 11 are satisfied for $k = R(e,i,6)$. Now, for all $k \in \{R(e,i,6), ..., M(e,i)\}$, q remains in critical section at γ_k. Hence, p is blocked by q and dis-

abled: the token remains at p all that time. This proves that the three items remains satisfied for all $k \in \{R(e,i,6), ..., M(e,i)\}$. □

Lemma 12. *Let $e = (\gamma_j)_{j\geq 0} \in \mathcal{E}$ be an execution of $\mathcal{LRA} \circ \mathcal{TC}$ and let $i \geq 0$ such that $\mathcal{TC}$ has stabilized at γ_i. We note $p \in V$ the unique token holder at γ_i, i.e., $TokenReady(\gamma_i(p))$ holds. If $R(e,i,4n)$ exists and $R(e,i,4n) \leq M(e,i)$, if $\forall j \in \{i, ..., R(e,i,4n)\}$, $TokenReady(\gamma_j(p))$ holds then for all $k \in \{R(e,i,4n), ..., M(e,i)\}$,*

$$P_{Free}(\gamma_k) \setminus \mathcal{CN}_p(\gamma_k) = \emptyset$$

Proof. Let $e = (\gamma_j)_{j\geq 0} \in \mathcal{E}$ be an execution of $\mathcal{LRA} \circ \mathcal{TC}$ and let $i \geq 0$ such that $\mathcal{TC}$ has stabilized at γ_i. We note $p \in V$ the unique token holder at γ_i. We assume that $R(e,i,4n)$ exists, $R(e,i,4n) \leq M(e,i)$ and $\forall j \in \{i, ..., R(e,i,4n)\}$, $TokenReady(\gamma_j(p))$.

P_{Free} contains requesting processes ($p.req \neq \perp$) with no neighbor q using a resource conflicting with the requested one (namely, such that $q.status = \mathsf{In}$ and $q.req \not\simeq p.req$). Note first that no process can enter P_{Free} since, no new request occurs and no critical section is released.

Let $j \in \{i, ..., R(e,i,4(n-1)\}$. If $P_{Free}(\gamma_j) \setminus \mathcal{CN}_p(\gamma_j)$ is empty, it remains so until $\gamma_{R(e,i,4n)}$. Otherwise, let $q = \max\{x \in P_{Free}(\gamma_j) \setminus \mathcal{CN}_p(\gamma_j)\}$. If q has status Out, it reaches status Wait in at most 2 rounds. Either q exited P_{Free} in the meantime, *i.e.*, a process with status Wait entered its critical section meanwhile and is using a conflicting resource, or q reaches status In in at most 2 additionnal rounds. Indeed, in the latter case, $IsBlocked(q)$ is false since $q \in P_{Free}$ ensures that $ResourceFree(q)$ and since it has no neighbor holding the token by assumption; furthermore, $q = Winner(q)$ by definition. Hence, at most 4 rounds later, q has exited P_{Free}.

Repeating the reasoning n times for $j = i$, $j = i+R(e,i,n)$, ..., $j = R(e,i,4(n-1)$ ensures that at configuration $\gamma_{R(e,i,4n)}$ the set $P_{Free}(\gamma_{R(e,i,4n)}) \setminus \mathcal{CN}_p(\gamma_{R(e,i,4n)})$ is empty. As long as no critical section is released, and as long as no new request occurs, it remains empty. □

Lemma 13. *Let $e = (\gamma_j)_{j\geq 0} \in \mathcal{E}$ be an execution of $\mathcal{LRA} \circ \mathcal{TC}$ and let $i \geq 0$ such that $\mathcal{TC}$ has stabilized at γ_i. If $R(e,i,6n(N_{tok}+1))$ exists and $R(e,i,6n(N_{tok}+1)) \leq M(e,i)$ then*

- *either for every $k \in \{R(e,i,6n(N_{tok}+1)), ..., M(e,i)\}$, $P_{Free}(\gamma_k) = \emptyset$*
- *or for every $k \in \{R(e,i,6n(N_{tok}+1)-6), ..., M(e,i)-1\}$, $PassToken$ is not executed at step $\gamma_k \mapsto \gamma_{k+1}$.*

Proof. Let $e = (\gamma_j)_{j\geq 0} \in \mathcal{E}$ be an execution of $\mathcal{LRA} \circ \mathcal{TC}$ and let $i \geq 0$ such that $\mathcal{TC}$ has stabilized at γ_i. Assume that $R(e,i,6n(N_{tok}+1))$ exists and $R(e,i,6n(N_{tok}+1)) \leq M(e,i)$.

Note again that P_{Free} cannot increase, hence if it is empty at some configuration γ_k with $k \in \{R(e,i,6n(N_{tok}+1)), ..., M(e,i)\}$, we are done.

Let $k \in \{R(e,i,6n(N_{tok}+1)), ..., M(e,i)\}$. Assume $P_{Free}(\gamma_k) \neq \emptyset$ and let $p \in P_{Free}(\gamma_k)$. We deduce from Lemma 11, that if $PassToken$ has not been executed by the token holder,

during 6 consecutive rounds, then the token will stay at this process until $M(e,i)$. Furthermore, properties of $\mathcal{TC}$ ensures that after at most N_{tok} executions of $PassToken$ the token will reach p. Then, at the latest, at configuration $\gamma_{R(e,k,6N_{tok})}$ ($6N_{tok}$ rounds later), the token is either blocked until $M(e,i)$ at some process (but not p) or has passed through p. Let consider the second case: if when the token is at p, P_{Free} still contains p, then, after at most 6 additionnal rounds (RsT-$action$, R-$action$, E-$action$), p has access to critical section and exits P_{Free}.

Repeating this reasoning n times, we have that in at most $6n(N_{tok}+1)$ rounds, either P_{Free} is empty or the token is blocked until $M(e,i)$. □

Lemma 14. *Algorithm $\mathcal{LRA} \circ \mathcal{TC}$ meets the **No Livelock** property of strong concurrency: there exists a number of rounds $T_{PC} > 0$ such that for every execution $e = (\gamma_i)_{i\geq 0} \in \mathcal{E}$ and for every index $i \geq 0$, if $R(e,i,T_{PC})$ exists then*

$$R(e,i,T_{PC}) \leq M(e,i) \Rightarrow \exists X, \mathcal{P}_{strong}(X, \gamma_{R(e,i,T_{PC})}) \wedge P_{Free}(\gamma_{R(e,i,T_{PC})}) \subseteq X$$

Proof. We pose $T_{PC} = T_{tok} + 6n(N_{tok}+1) + 4n$. Let $e = (\gamma_i)_{i\geq 0} \in \mathcal{E}$ be an execution of $\mathcal{LRA} \circ \mathcal{TC}$ and let $i \geq 0$. Assume that $R(e,i,T_{PC})$ exists and $R(e,i,T_{PC}) \leq M(e,i)$. After T_{tok} rounds, $\mathcal{TC}$ has stabilized. Using Lemma 13, we have two cases.

First case: after $T_{tok} + 6n(N_{tok}+1)$, P_{Free} is empty and remains so until $M(e,i)$. In this case, we are done.

Second case: for every $k \in \{R(e,i,6n(N_{tok}+1)-6), ..., M(e,i)-1\}$, $PassToken$ is not executed at step $\gamma_k \mapsto \gamma_{k+1}$. Note that this implies that $PassToken$ is not executed during the last 6 rounds by the token holder, say p: this allows to apply Lemma 11 and to show that there exists a conflicting neighbor of p, q, such that $\forall r \in P_{Free} \cap \mathcal{CN}_p(\gamma_k)$, $r \notin \{q\} \cup \mathcal{CN}_q(\gamma_k)$.

As p holds the token from configuration $R(e,i,6n(N_{tok}+1)-6)$ to configuration $M(e,i)$, and as $R(e,i,T_{tok}+6n(N_{tok}+1)+4n) \leq M(e,i)$, we can apply Lemma 12 between configuration $R(e,i,T_{tok}+6n(N_{tok}+1)$ and $R(e,i,T_{tok}+6n(N_{tok}+1)+4n)$: this proves that $P_{Free}(\gamma_{R(e,i,T_{tok}+6n(N_{tok}+1)+4n)}) \setminus \mathcal{CN}_p(\gamma_{R(e,i,T_{tok}+6n(N_{tok}+1)+4n)})$ is empty. This concludes the proof. □

Lemma 10 and 14 proves strong concurrency of $\mathcal{LRA} \circ \mathcal{TC}$.

Theorem 7. *Algorithm $\mathcal{LRA} \circ \mathcal{TC}$ is strongly concurrent.*

9 Conclusion

We characterized the maximal level of concurrency we can obtain in resource allocation problems by proposing the notion of *maximal-concurrency*. This notion is versatile, *e.g.*, it generalizes the avoiding ℓ-deadlock [19] and (strict) (k,ℓ)-liveness [10] defined for the ℓ-exclusion and

k-out-of-ℓ-exclusion, respectively. From [19], we already know that *maximal-concurrency* can be achieved in some important global resource allocation problems.[3] Now, perhaps surprisingly, our results show that *maximal-concurrency* cannot be achieved in problems that can be expressed with the LRA paradigm. However, we showed that *strong partial maximal-concurrency* (an high, but not maximal, level of concurrency) can be achieved by a snap-stabilizing LRA algorithm. We have to underline that the level of concurrency we achieve here is similar to the one obtained in the committee coordination problem [5]. Defining the exact class of resource allocation problems where *maximal-concurrency* (resp. *strong partial maximal-concurrency*) can be achieved is a challenging perspective.

References

[1] Karine Altisen, Alain Cournier, Stéphane Devismes, Anaïs Durand, and Franck Petit. Self-stabilizing Leader Election in Polynomial Steps. In *Stabilization, Safety, and Security of Distributed Systems - 16th International Symposium, SSS 2014, Paderborn, Germany, September 28 - October 1, 2014. Proceedings*, pages 106–119, 2014.

[2] Karine Altisen and Stéphane Devismes. On Probabilistic Snap-Stabilization. In *ICDCN'2014, 15th International Conference on Distributed Computing and Networking*, pages 272–286, Coimbatore, India, January 4-7 2014. LNCS.

[3] Anish Arora and Mohamed G. Gouda. Distributed Reset. *IEEE Trans. Computers*, 43(9):1026–1038, 1994.

[4] Joffroy Beauquier, Ajoy Kumar Datta, Maria Gradinariu, and Frédéric Magniette. Self-Stabilizing Local Mutual Exclusion and Daemon Refinement. *Chicago J. Theor. Comput. Sci.*, 2002, 2002.

[5] Borzoo Bonakdarpour, Stéphane Devismes, and Franck Petit. Snap-Stabilizing Comittee Coordination. In *25th IEEE International Symposium on Parallel and Distributed Processing IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*, pages 231–242, 2011.

[6] Christian Boulinier, Franck Petit, and Vincent Villain. When Graph Theory Helps Self-Stabilization. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004*, pages 150–159, 2004.

[3] By "global" we mean resource allocation problems where a resource can be accessed by any process.

[7] Alain Bui, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. Snap-Stabilization and PIF in Tree Networks. *Distributed Computing*, 20(1):3–19, 2007.

[8] Sébastien Cantarell, Ajoy Kumar Datta, and Franck Petit. Self-Stabilizing Atomicity Refinement Allowing Neighborhood Concurrency. In *Self-Stabilizing Systems, 6th International Symposium, SSS 2003, San Francisco, CA, USA, June 24-25, 2003, Proceedings*, pages 102–112, 2003.

[9] Alain Cournier, Stéphane Devismes, and Vincent Villain. Light Enabling Snap-Stabilization of Fundamental Protocols. *TAAS*, 4(1), 2009.

[10] Ajoy Kumar Datta, Rachid Hadid, and Vincent Villain. A Self-Stabilizing Token-Based k-out-of-l-Exclusion Algorithm. *Concurrency and Computation: Practice and Experience*, 15(11-12):1069–1091, 2003.

[11] Ajoy Kumar Datta, Rachid Hadid, and Vincent Villain. A new self-stabilizing k-out-of-l exclusion algorithm on rings. In *Self-Stabilizing Systems, 6th International Symposium, SSS 2003, San Francisco, CA, USA, June 24-25, 2003, Proceedings*, pages 113–128, 2003.

[12] Ajoy Kumar Datta, Colette Johnen, Franck Petit, and Vincent Villain. Self-Stabilizing Depth-First Token Circulation in Arbitrary Rooted Networks. *Distributed Computing*, 13(4):207–218, 2000.

[13] Ajoy Kumar Datta, Lawrence L. Larmore, and Priyanka Vemula. Self-stabilizing Leader Election in Optimal Space under an Arbitrary Scheduler. *Theor. Comput. Sci.*, 412(40):5541–5561, 2011.

[14] Edsger W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Commun. ACM*, 8(9):569, 1965.

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

[16] Edsger W Dijkstra. Two Starvation-Free Solutions of a General Exclusion Problem. Technical Report EWD 625, Plataanstraat 5, 5671, AL Nuenen, The Netherlands, 1978.

[17] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

[18] Shlomi Dolev and Ted Herman. Superstabilizing Protocols for Dynamic Distributed Systems. *Chicago J. Theor. Comput. Sci.*, 1997, 1997.

[19] Michael J. Fischer, Nancy A. Lynch, James E. Burns, and Allan Borodin. Resource Allocation with Immunity to Limited Process Failure (Preliminary Report). In *20th Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 29-31 October 1979*, pages 234–254, 1979.

[20] Mohamed G. Gouda and F. Furman Haddix. The Alternator. *Distributed Computing*, 20(1):21–28, 2007.

[21] Maria Gradinariu and Sébastien Tixeuil. Conflict managers for self-stabilization without fairness assumption. In *27th IEEE International Conference on Distributed Computing Systems (ICDCS 2007), June 25-29, 2007, Toronto, Ontario, Canada*, page 46, 2007.

[22] Shing-Tsaan Huang. The Fuzzy Philosophers. In *Parallel and Distributed Processing, 15 IPDPS 2000 Workshops, Cancun, Mexico, May 1-5, 2000, Proceedings*, pages 130–136, 2000.

[23] Shing-Tsaan Huang and Nian-Shing Chen. Self-Stabilizing Depth-First Token Circulation on Networks. *Distributed Computing*, 7(1):61–66, 1993.

[24] Hirotsugu Kakugawa and Masafumi Yamashita. Self-Stabilizing Local Mutual Exclusion on Networks in which Process Identifiers are not Distinct. In *21st Symposium on Reliable Distributed Systems (SRDS 2002), 13-16 October 2002, Osaka, Japan*, pages 202–211, 2002.

[25] Leslie Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Commun. ACM*, 17(8):453–455, 1974.

[26] Mikhail Nesterenko and Anish Arora. Stabilization-Preserving Atomicity Refinement. *J. Parallel Distrib. Comput.*, 62(5):766–791, 2002.

[27] Michel Raynal. A Distributed Solution to the k-out of-M Resources Allocation Problem. In *Advances in Computing and Information - ICCI'91, International Conference on Computing and Information, Ottawa, Canada, May 27-29, 1991, Proceedings*, pages 599–609, 1991.