# 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-concurrency*.

## 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 $\ell$-*exclusion* problem [19] is a generalization of the mutual exclusion problem which allows use of $\ell$ 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-*$\ell$-*exclusion* problem [27] is a generalization of $\ell$-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.*, [5, 7, 9, 20, 22, 24, 26]

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

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 $\ell$-exclusion. To address this issue, Fischer *et al.* [19] proposed an *ad hoc* property to capture concurrency in $\ell$-exclusion. This property is called *avoiding $\ell$-deadlock* and is informally defined as follows: "if fewer than $\ell$ 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 $\ell$-deadlock property, have been proposed to capture the level of concurrency in other resource allocation problems, *e.g.*, $k$-out-of-$\ell$-exclusion [11] and committee coordination [6]. However, until now, all existing properties of concurrency are specific to a particular problem, *e.g.*, the avoiding $\ell$-deadlock property cannot be applied to committee coordination.

In this paper, we first propose to generalize the definition of avoiding $\ell$-deadlock to any resource allocation problems. We call this new property the *maximal-concurrency*. Then, we consider the maximal-concurrency in the context of the *Local Resource Allocation (LRA)* problem, defined by Cantarell *et al.* [9]. 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 $\ell$-exclusion and local $k$-out-of-$\ell$-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-concurrency* this weaker version of maximal concurrency. The goal of *partial-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-concurrency in bidirectional identified networks of arbitrary topology. As additional feature, this algorithm is *snap-stabilizing* [8]. *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.*, [5, 7, 9, 20, 22, 24, 26]. In [7], 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.

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

There are also many self-stabilizing algorithms for local mutual exclusion [5, 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 [9], 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 [9]) 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.* Next section introduces the computation model and the specification of the LRA problem. In Section 3, we define the property of maximal-concurrency, show the impossibility result, and then circumvent this impossibility by introducing the partial-concurrency. Our algorithm is presented in Section 4. We outline the proofs of its correctness and (strong) partial-concurrency in Subsection 4.4. A detailed proof is available in the technical report [3]. We conclude in Section 5.

## 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, 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 $\gamma$ of the system is the vector of states of all processes. We denote by $\gamma(p)$ the state of a process $p$ in a configuration $\gamma$.

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: $(\langle\text{priority}\rangle)$ $\langle\text{label}\rangle : \langle\text{guard}\rangle \rightarrow \langle\text{statement}\rangle$. 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 $\gamma$. 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 $\gamma$, 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 $\gamma$, leading the system to a new configuration $\gamma'$. We denote by $\mapsto$ the relation between configurations such that $\gamma \mapsto \gamma'$ if and only if $\gamma'$ can be reached from $\gamma$ 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 $\gamma$ 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.

### 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 $p$.

**Local Resource Allocation.** The *Local Resource Allocation (LRA)* problem [9] 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 $X \not\rightleftharpoons Y$.

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 $X \neq X$.

*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 $W \neq R$ and $W \neq W$.

**Snap-Stabilization.** Let $\mathcal{A}$ be a distributed algorithm. A *specification* $SP$ is a predicate over all executions of $\mathcal{A}$. In [8], 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 $\gamma_i$. 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 $\gamma_i \mapsto \gamma_{i+1}$.
- $Result(\gamma_i \ldots \gamma_j, p, r)$ 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 $\gamma_i$ and $\gamma_j$.
- $NoConflict(\gamma_i, p)$ means that, in $\gamma_i$, 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:* $\forall k \geq 0, \forall k' > k, \forall p \in V, \forall r \in \mathcal{R}_p, \big[Result(\gamma_k \ldots \gamma_{k'}, p, r) \wedge \big(\exists l < k, Start(\gamma_l, \gamma_{l+1}, p, r)\big)\big] \Rightarrow \big[\forall i \in \{k, \ldots, k'\}, NoConflict(\gamma_i, p)\big]$

**Computation Start:** *If an application at process $p$ requests resource $r$, then $p$ eventually starts a computation to obtain $r$:* $\forall k \geq 0, \forall p \in V, \forall r \in \mathcal{R}_p, \big[\exists l > k, Request(\gamma_l, p, r) \Rightarrow Start(\gamma_l, \gamma_{l+1}, p, r)\big]$

**Computation End:** *If process $p$ starts a computation to obtain resource $r$, the computation eventually ends (in particular, $p$ obtained $r$ during the computation):* $\forall k \geq 0$, $\forall p \in V, \forall r \in \mathcal{R}_p$, $Start(\gamma_k, \gamma_{k+1}, p, r) \Rightarrow \big[\exists l > k, \exists l' > l, Result(\gamma_l \dots \gamma_{l'}, p, r)\big]$

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 Concurrency

Many existing resource allocation algorithms, especially self-stabilizing ones [5, 7, 9, 20, 22, 24, 26], do not consider the concurrency issue. In [19], authors propose a concurrency property *ad hoc* to $\ell$-exclusion. We now define the *maximal-concurrency*, which generalizes the definition of [19] to any resource allocation problem.

### 3.1 Maximal-Concurrency.

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 $\gamma$, *i.e.*, the set of processes holding resources in $\gamma$. Let $P_{Req}(\gamma)$ be the set of requesting processes that are not in critical section in $\gamma$. Let $P_{Free}(\gamma) \subseteq P_{req}(\gamma)$ be the set of requesting processes that can access their requested resource(s) in $\gamma$ without violating the safety of the considered resource allocation problem. Let $continuousCS(\gamma_i \dots \gamma_j) \equiv \forall k \in \{i, \dots, j-1\}, P_{CS}(\gamma_k) \subseteq P_{CS}(\gamma_{k+1})$

**Definition 1 (Maximal-Concurrency).** *An algorithm is* maximal-concurrent *if and only if* $\forall e = (\gamma_i)_{i \geq 0} \in \mathcal{E}$, $\forall i \geq 0$, $\exists N \in \mathbb{N}$, $\forall j > N$, $\big(continuousCS(\gamma_i \dots \gamma_{i+j}) \wedge P_{Free}(\gamma_i) \neq \emptyset\big) \Rightarrow \big(\exists k \in \{i, \dots, i+j-1\}, \exists p \in V, p \in P_{Free}(\gamma_k) \cap P_{CS}(\gamma_{k+1})\big)$

The examples below show the versatility of our property: we instantiate the set $P_{Free}$ according to the considered problem.

*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 $\gamma$. Hence,

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

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

*Example 2: $\ell$-Exclusion Maximal-Concurrency.* The $\ell$-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 $\ell$ 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 $\ell$-deadlock" property of Fischer *et al.* [19].

*Example 3: $k$-out-of-$\ell$ Exclusion Maximal-Concurrency.* The $k$-out-of-$\ell$ exclusion problem is a generalization of the $\ell$-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 $(k, \ell)$-liveness" property of Datta *et al.* [**?**], 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-$\ell$ exclusion algorithm satisfying the strict $(k, \ell)$-liveness. To circumvent this impossibility, they then propose a weaker property called "$(k, \ell)$-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.

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

**Definition 2 (Maximal Concurrency).** *An algorithm is* maximal concurrent *if and only if* $\forall e = (\gamma_i)_{i \geq 0} \in \mathcal{E}$, $\forall i \geq 0$, $\exists T \in \mathbb{N}$, $\forall t \geq T$,

$$continuousCS(\gamma_i \dots \gamma_{i+t}) \Rightarrow P_{Free}(\gamma_{i+t}) = \emptyset$$

Definitions 1 and 2 are equivalent using induction arguments (see [3]). Using the latter definition, remark that an algorithm is not maximal concurrent if and only if $\exists e = (\gamma_i)_{i \geq 0} \in \mathcal{E}, \exists i \geq 0, \forall T \in \mathbb{N}, \exists t \geq T, continuousCS(\gamma_i \dots \gamma_{i+t}) \wedge P_{Free}(\gamma_{i+t}) \neq \emptyset$.

### 3.2 Maximal Concurrency vs. Fairness.

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, Theorem 1 states that no LRA algorithm (stabilizing or not) can achieve maximal-concurrency. Actually, its proof is based on the incompatibility between fairness and maximal-concurrency.

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

**Theorem 1.** *It is impossible to design a LRA algorithm for arbitrary networks that satisfies maximal-concurrency.*

*Proof.* Assume, by contradiction, that there is a local resource allocation algorithm $\mathcal{A}$ (stabilizing or not) which satisfies maximal-concurrency. Let consider the following graph: $G = (V, E)$ where $V = \{p_1, p_2, p_3\}$ and $E = \{(p_1, p_2), (p_2, p_3)\}$. Let $X$ and $Y$ be two resources such that $X \neq Y$, $X \in \mathcal{R}_{p_1}$, $Y \in \mathcal{R}_{p_2}$, and $X \in \mathcal{R}_{p_3}$ (notice that we can have $X = Y$). We assume that, when $p_1$ and $p_3$ request a resource, they request $X$, and, when $p_2$ requests a resource, it requests $Y$. Below, we exhibit a possible execution $e$ of $\mathcal{A}$ on $G$ where fairness is violated if maximal-concurrency is achieved. Figure 1 illustrates the proof.

First, assume that $p_1$ continuously requests $X$ while $p_2$ and $p_3$ are idle (Configuration 1.(a)). As $\mathcal{A}$ satisfies the fairness property, $p_1$ eventually executes its critical section to access $X$. This critical section can last an arbitrary long (yet finite) time (Figure 1.(b)).

Then, $p_2$ and $p_3$ start continuously requesting ($Y$ for $p_2$ and $X$ for $p_3$). To satisfy the maximal-concurrency property, $p_3$ must eventually obtain resource $X$, even if $p_1$ does not finish its critical section in the meantime. In this case, the system reaches the configuration given in Figure 1.(d).

Then, it is possible that $p_1$ ends its critical section and releases resource $X$ right after Configuration 1.(d). But, in this case, $p_2$ still cannot access $Y$ because $Y$ is conflicting with the resource $X$ currently used by $p_3$. So, the system can reach Configuration 1.(e). If $p_1$ continuously requests $X$ again right after Configuration 1.(e), we obtain Configuration 1.(f). Now, the execution of the critical section of $p_3$ may last an arbitrary long (yet finite) time, and $p_1$ should again access $X$, even if $p_3$ does not finish its critical section in the meantime, by maximal-concurrency. So, the system can reach Configuration 1.(g).

Now, if $p_3$ releases its resource and then continuously requests it again, we retrieve a configuration similar to the one of Figure 1.(c). We can repeat this scheme infinitely often so that $p_2$ continuously requests $Y$ but never access it: the fairness property is violated, a contradiction. □

### 3.3 Partial Maximal-Concurrency.

To circumvent the previous impossibility result, we propose a weaker version of maximal concurrency, called *partial maximal-concurrency*.

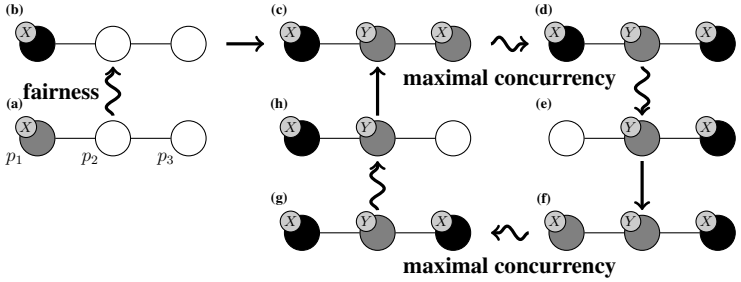



**Fig. 1.** Maximal concurrency *vs.* fairness. The processes in black are executing their critical section. The processes in gray are requesting resources. The processes in white are idle. Requested resources are given in the bubbles next to the nodes.

**Definition 4 (Partial Maximal-Concurrency).** *An algorithm $\mathcal{A}$ is* partially maximal-concurrent *if and only if* $\forall e = (\gamma_i)_{i\geq 0} \in \mathcal{E}$, $\forall i \geq 0$, $\exists T \in \mathbb{N}$ *such that* $\forall t \geq T$, $\exists X \subseteq V$ *such that* $continuousCS(\gamma_i \dots \gamma_{i+t}) \Rightarrow P_{Free}(\gamma_{i+t}) \subseteq X$

Notice that, by definition, a maximal-concurrent algorithm is also partially maximal-concurrent.

The proof of Theorem 1 reveals that fairness and maximal concurrency are contradictory in the following situation: some neighbors of a process alternatively use resources which are conflicting with its own request. So, to achieve fairness, we must relax the expected level of concurrency in such a way that at least in that situation $p$ eventually satisfies its request. To ensure this, any LRA algorithm should then eventually allow $p$ to prevent its requesting neighbors from entering their critical section, even if $p$ cannot currently satisfies its request (*i.e.*, even if one of its neighbor is using a conflicting resource) and even if some of its requesting neighbors can enter critical section without creating any conflict. Hence, in the worst case, $p$ has one neighbor holding a conflicting resource and it should prevent all other neighbors to satisfy their requests, in order to eventually satisfy its own request (and so to ensure fairness).

We derive the following refinement of partial maximal concurrency based on this latter observation: this seems to be the finest concurrency we can expect in LRA algorithm.

**Definition 5 (Strong Partial Maximal-Concurrency).** *An algorithm $\mathcal{A}$ is* strongly partially maximal-concurrent *if and only if* $\forall e = (\gamma_i)_{i\geq 0} \in \mathcal{E}$, $\forall i \geq 0$, $\exists T \in \mathbb{N}$ *such that* $\forall t \geq T$, $\exists p, q \in V$, $q \in \mathcal{N}_p$ *such that* $continuousCS(\gamma_i \dots \gamma_{i+t}) \Rightarrow P_{Free}(\gamma_{i+t}) \subseteq \mathcal{N}_p \backslash \{q\}$

In the next section, we show that strong partial maximal-concurrency can be realized by a snap-stabilizing LRA algorithm.

## 4 Local Resource Allocation Algorithm

We now propose a snap-stabilizing LRA algorithm which achieves the strong partial maximal 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.

### 4.1 Composition

These two modules are composed using a *fair composition* [17], denoted $\mathcal{LRA} \circ \mathcal{TC}$. 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. Such a translation can be done using the rewriting rule given in the technical report [3].

### 4.2 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 whether 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 [10, 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 [4, 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 1.* 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.

**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**
$p.req \in \mathcal{R}_p \cup \{\bot\}$: Variable from the application
$TokenReady(p)$: Predicate from $\mathcal{TC}$, indicate that p holds the token
$PassToken(p)$: Statement from $\mathcal{TC}$, pass the token to a neighbor
**Macros**
$WaitingNeigh(p) \equiv \{q \in \mathcal{N}_p \mid q.status = \mathsf{Wait}\}$
$LocalMax(p) \equiv \max\{q \in WaitingNeigh(p) \cup \{p\}\}$
$LocalTokens(p) \equiv \{q \in \mathcal{N}_p \cup \{p\} \mid q.token\}$
$TokenMax(p) \equiv \max\{q \in LocalTokens(p)\}$
**Predicates**
$ResourceFree(p) \equiv \forall q \in \mathcal{N}_p, (q.status = \mathsf{In} \Rightarrow p.req \rightleftharpoons q.req)$
$IsBlocked(p) \equiv \neg ResourceFree(p) \vee (\exists q \in \mathcal{N}_p, q.status = \mathsf{Blocked} \wedge q.token)$
$TokenAccess(p) \equiv LocalTokens(p) \neq \emptyset \wedge p = TokenMax(p)$
$MaxAccess(p) \equiv LocalTokens(p) = \emptyset \wedge p = LocalMax(p)$
**Guards**
$Requested(p) \equiv p.status = \mathsf{Out} \wedge p.req \neq \bot$
$Block(p) \equiv p.status = \mathsf{Wait} \wedge IsBlocked(p)$
$Unblock(p) \equiv p.status = \mathsf{Blocked} \wedge \neg IsBlocked(p)$
$Enter(p) \equiv p.status = \mathsf{Wait} \wedge \neg IsBlocked(p) \wedge (TokenAccess(p) \vee MaxAccess(p))$
$Exit(p) \equiv p.status = \mathsf{In} \wedge p.req = \bot$
$ResetToken(p) \equiv TokenReady(p) \neq p.token$
$ReleaseToken(p) \equiv TokenReady(p) \wedge p.status \in \{\mathsf{Out}, \mathsf{In}\} \wedge \neg Requested(p)$
**Actions**
(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) $E\text{-}action :: Enter(p) \rightarrow p.status \leftarrow \mathsf{In}$; **if** $TokenReady(p)$ **then** $PassToken(p)$ **fi**;

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

### 4.3 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: $p.req \in \mathcal{R}_p \cup \{\bot\}$ 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 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 $p.req = \bot$, 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:

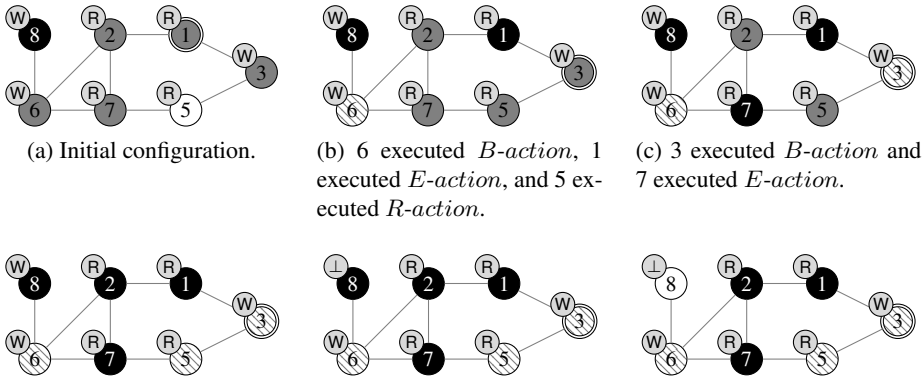
- The value of $p.req$ can change from $\perp$ to $r \in \mathcal{R}_p$ if and only if $p.status = \mathsf{Out}$,
- The value of $p.req$ can change from $r \in \mathcal{R}_p$ to $\perp$ if and only if $p.status = \mathsf{In}$.
- The value of $p.req$ cannot directly change from $r \in \mathcal{R}_p$ to $r' \in \mathcal{R}_p$ with $r' \neq r$.

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

Consequently, the predicate $Request(\gamma_i, p, r)$ in Specification 1 is true if and only if $p.req = r$ in $\gamma_i$; the predicate $NoConflict(\gamma_i, p)$ is expressed by $p.status = \mathsf{In} \Rightarrow \big(\forall q \in \mathcal{N}_p, q.status = \mathsf{In} \Rightarrow (q.req \rightleftharpoons p.req)\big)$ in $\gamma_i$. (We set $\perp$ 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 $p.req = r \neq \perp$ in $\gamma_i$.

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 $\gamma_j \mapsto \gamma_{j+1}$.



(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 $Ex\text{-}action$.

**Fig. 2.** Example of execution of $\mathcal{LRA} \circ \mathcal{TC}$.

We now illustrate the principles of $\mathcal{LRA}$ with the example given in Figure 2. 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.

When the process is idle ($p.status = \mathsf{Out}$), its application can request a resource. In this case, $p.req = r \neq \perp$ and $p$ sets $p.status$ to Wait by $R$-$action$: $p$ starts the computation to obtain $r$. 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 token in the neighborhood of $p$, the highest priority process is the waiting process with highest ID. Otherwise, the token holders (there may be several tokens during the stabilization phase of $\mathcal{TC}$) blocked all their requesting neighbors, even if they request 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.

### 4.4 Correctness and Partial Maximal-Concurrency

In this subsection, we sketch the proof of snap-stabilization of Algorithm $\mathcal{LRA} \circ \mathcal{TC}$. Then, we give the proof outline which shows that $\mathcal{LRA} \circ \mathcal{TC}$ is strongly partially maximal-concurrent. Recall that we assume a distributed weakly fair daemon.

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

*Proof Outline.* Immediate from the guard of $E$-$action$. □

In $\mathcal{LRA} \circ \mathcal{TC}$, the token circulation is used to ensure fairness. Hence, a crucial point to show that $\mathcal{LRA} \circ \mathcal{TC}$ satisfies the computation start and end properties (Theorems 3 and 4) consists in showing that no process can keep a token forever.

**Lemma 1.** *No process can keep a token forever.*

*Proof Outline.* Assume, by contradiction, that a process $p$ holds a token forever. Then, eventually $p$ is the only token holder forever, by Property 1. If $p.status = \mathsf{Out}$ and $p.req = \bot$, $p$ does not need the token and straightaway releases it by $RlT\text{-}action$, a contradiction. If $p.status = \mathsf{In}$, $p$ eventually ends its computation executing $Ex\text{-}action$. Then, $p.status = \mathsf{Out}$, and, as in the previous case, $p$ eventually releases its token, a contradiction. Otherwise, the token gives priority to $p$ over all of its neighbors. So, $p$ eventually enters in critical section by $E\text{-}action$ and so releases the token, a contradiction. □

**Theorem 3 (Computation End).** *Every execution of $\mathcal{LRA} \circ \mathcal{TC}$ satisfies the computation end property.*

*Proof Outline.* Assume a computation starts at process $p$ to obtain resource $r$.

Assume, by contradiction, that $r$ is never allocated to $p$. By Property 1, a unique token eventually exists in the network. Moreover, $p$ eventually gets the token, by Lemma 1 and Property 3. Again by Lemma 1, $p$ eventually releases the token. Now, $p$ can only release the token by executing $E\text{-}action$. In this case, $p$ obtains resource $r$, a contradiction.

Hence, $r$ is allocated to $p$ in finite time. Now, by Property 5, in finite time, the application does not need the resource $r$ anymore and sets $p.req$ to $\bot$. So $p$ eventually executes $Ex\text{-}action$ and ends its computation. □

We illustrate the previous proof with an example given in Figure 3. We consider the local mutual exclusion problem. In this example, we try to delay as much as possible the critical section of process 2. First, process 2 has two neighbors (7 and 8) that also request the resource and have greater IDs. So, they will execute their critical section before 2 (in steps (a)↦(b) and (e)↦(f)). But, the token circulates and eventually reaches 2 (see Configuration (g)). Then, 2 has priority over its neighbors (even though it has a lower ID) and eventually starts executing its critical section in (j)↦(k)).

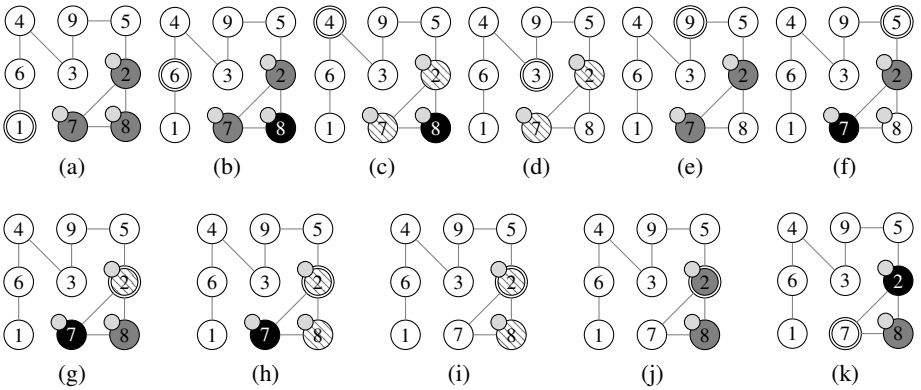



**Fig. 3.** Example of execution of $\mathcal{LRA} \circ \mathcal{TC}$ on the local mutual exclusion problem. The bubbles mark the requesting processes.

**Theorem 4 (Computation Start).** *Every execution of $\mathcal{LRA} \circ \mathcal{TC}$ satisfies the computation start property.*

*Proof Outline.* A process $p$ eventually obtains status Out. Indeed, if $p.status \neq \mathsf{Out}$, $p$ is computing and, by Theorem 3, this computation eventually ends. Hence, if the application of $p$ requests some resource $r$, *i.e.*, $p.req = r \neq \perp$, $p$ eventually executes $R\text{-}action$ and a computation for $r$ starts. □

Theorem 5 below is immediate from Theorems 2, 3, and 4.

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

We now show that $\mathcal{LRA} \circ \mathcal{TC}$ is strongly partially maximal-concurrent. We instantiate the sets $P_{CS}$ and $P_{Req}$ as follows: $P_{Req}(\gamma) = \{p \in V, p.req \neq \perp \wedge p.status \neq \mathsf{In} \text{ in } \gamma\}$ and $P_{CS}(\gamma) = \{p \in V, p.status = \mathsf{In} \wedge p.req \neq \perp \text{ in } \gamma\}$.

**Theorem 6 (Strong Partial Maximal-Concurrency).** *Algorithm $\mathcal{LRA} \circ \mathcal{TC}$ is a strong partial maximal concurrent local resource allocation algorithm.*

*Proof Outline.* After stabilization of $\mathcal{TC}$, $\exists T$ from which, if $continousCS$ holds until $\gamma_T$, then every process does not change the values of its variables $req$ and $status$. After $\gamma_T$ (and if $continuousCS$ still holds), if $P_{Free}$ is not empty, every process in $P_{Free}$ has status Blocked. Indeed, otherwise there is a finite sequence of processes in $P_{Free}$ with increasing priorities such that the last process is allowed to execute $E\text{-}action$ and change its $status$ to In, a contradiction with the definition of $T$.

A process $p$ is blocked because $\neg ResourceFree(p)$ or $\big(\exists q \in \mathcal{N}_p, q.status = \mathsf{Blocked} \wedge q.token\big)$. Now, in the former case, $p \notin P_{Free}$. So, $p \in P_{Free}$ is blocked because of the unique token holder, say $q$. Then, $p \in \mathcal{N}_q$ and $P_{Free}(\gamma_T)$ contains all the requesting neighbors of $q$. In the worst case, it contains all the neighborhood of $q$ except one process $s$ that is in critical section, namely, the one that blocks $q$. Hence, $P_{Free}(\gamma_T) \subseteq \mathcal{N}_q \backslash \{s\}$, and $\mathcal{LRA} \circ \mathcal{TC}$ is strongly partially maximal-concurrent. □

## 5 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 $\ell$-deadlock [19] and (strict) $(k,\ell)$-liveness [11] defined for the $\ell$-exclusion and $k$-out-of-$\ell$-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 [6]. Defining the exact class of resource allocation problems where *maximal-concurrency* (resp. *strong partial maximal-concurrency*) can ~~be achieved is~~ a challenging perspective.

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

## References


1. Altisen, K., Cournier, A., Devismes, S., Durand, A., Petit, F.: Self-stabilizing Leader Election in Polynomial Steps. In: SSS. pp. 106–119 (2014)
2. Altisen, K., Devismes, S.: On Probabilistic Snap-Stabilization. In: ICDCN. pp. 272–286 (2014)
3. Altisen, K., Devismes, S., Durand, A.: Concurrency in Snap-Stabilizing Local Resource Allocation. Research report, VERIMAG (Dec 2014), hal.archives-ouvertes.fr/hal-01099186
4. Arora, A., Gouda, M.G.: Distributed Reset. IEEE Trans. Computers 43(9), 1026–1038 (1994)
5. Beauquier, J., Datta, A.K., Gradinariu, M., Magniette, F.: Self-Stabilizing Local Mutual Exclusion and Daemon Refinement. Chicago J. Theor. Comput. Sci. (2002)
6. Bonakdarpour, B., Devismes, S., Petit, F.: Snap-Stabilizing Comittee Coordination. In: IPDPS. pp. 231–242 (2011)
7. Boulinier, C., Petit, F., Villain, V.: When Graph Theory Helps Self-Stabilization. In: PODC. pp. 150–159 (2004)
8. Bui, A., Datta, A.K., Petit, F., Villain, V.: Snap-Stabilization and PIF in Tree Networks. Dist. Comp. 20(1), 3–19 (2007)
9. Cantarell, S., Datta, A.K., Petit, F.: Self-Stabilizing Atomicity Refinement Allowing Neighborhood Concurrency. In: SSS. pp. 102–112 (2003)
10. Cournier, A., Devismes, S., Villain, V.: Light Enabling Snap-Stabilization of Fundamental Protocols. ACM TAAS 4(1) (2009)
11. Datta, A.K., Hadid, R., Villain, V.: A Self-Stabilizing Token-Based k-out-of-l-Exclusion Algorithm. Concurrency and Computation: Practice and Experience 15(11-12), 1069–1091 (2003)
12. Datta, A.K., Johnen, C., Petit, F., Villain, V.: Self-Stabilizing Depth-First Token Circulation in Arbitrary Rooted Networks. Dist. Comp. 13(4), 207–218 (2000)
13. Datta, A.K., Larmore, L.L., Vemula, P.: Self-stabilizing Leader Election in Optimal Space under an Arbitrary Scheduler. Theor. Comput. Sci. 412(40), 5541–5561 (2011)
14. Dijkstra, E.W.: Solution of a Problem in Concurrent Programming Control. Commun. ACM 8(9), 569 (1965)
15. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM 17(11), 643–644 (1974)
16. Dijkstra, E.W.: Two Starvation-Free Solutions of a General Exclusion Problem. Tech. Rep. EWD 625, Plataanstraat 5, 5671, AL Nuenen, The Netherlands (1978)
17. Dolev, S.: Self-Stabilization. MIT Press (2000)
18. Dolev, S., Herman, T.: Superstabilizing Protocols for Dynamic Distributed Systems. Chicago J. Theor. Comput. Sci. (1997)
19. Fischer, M.J., Lynch, N.A., Burns, J.E., Borodin, A.: Resource Allocation with Immunity to Limited Process Failure (Preliminary Report). In: FOCS. pp. 234–254 (1979)
20. Gouda, M.G., Haddix, F.F.: The Alternator. Dist. Comp. 20(1), 21–28 (2007)
21. Gradinariu, M., Tixeuil, S.: Conflict Managers for Self-Stabilization without Fairness Assumption. In: ICDCS. p. 46 (2007)
22. Huang, S.: The Fuzzy Philosophers. In: IPDPS. pp. 130–136 (2000)
23. Huang, S., Chen, N.: Self-Stabilizing Depth-First Token Circulation on Networks. Dist. Comp. 7(1), 61–66 (1993)
24. Kakugawa, H., Yamashita, M.: Self-Stabilizing Local Mutual Exclusion on Networks in which Process Identifiers are not Distinct. In: SRDS. pp. 202–211 (2002)
25. Lamport, L.: A New Solution of Dijkstra's Concurrent Programming Problem. Commun. ACM 17(8), 453–455 (1974)

26. Nesterenko, M., Arora, A.: Stabilization-Preserving Atomicity Refinement. J. Parallel Distrib. Comput. 62(5), 766–791 (2002)
27. Raynal, M.: A Distributed Solution to the k-out of-M Resources Allocation Problem. In: ICCI'91. pp. 599–609 (1991)