Concurrency in Snap-Stabilizing Local Resource Allocation *

Karine Altisen, Stéphane Devismes, and Anaïs Durand

VERIMAG UMR 5104
Université Grenoble Alpes, France

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 there is a wide class of instances of LRA for which it is impossible to obtain maximal concurrency 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*.

Keywords: Distributed algorithms, snap-stabilization, self-stabilization, local resource allocation.

1 Introduction

Mutual exclusion [13, 24] 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 [26] 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 [15], two neighbors cannot use their common resource simultaneously.

*This work has been partially supported by the ANR projects DESCARTES (ANR-16-CE40-0023) and ESTATE (ANR-16-CE25-0009).

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 resource allocation problems are specified in terms of safety and liveness properties only, *i.e.*, most of them include no property addressing concurrency performances, *e.g.*, [5, 7, 20, 22, 25].

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 [9] and committee coordination [4]. However, until now, all existing properties of concurrency are specific to a particular problem, *e.g.*, the avoiding ℓ-deadlock property cannot be applied to committee coordination.

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 consider the maximal concurrency in the context of the *Local Resource Allocation (LRA)* problem, defined by Cantarell *et al.* [7]. LRA is a generalization of resource allocation problems in which resources are shared among neighboring processes. Dining philosophers, local readers-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.

We show that maximal concurrency cannot be achieved in a wide class of instances of the LRA problem. 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 together. 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 any instance of the LRA problem without compromising fairness.

We propose an LRA algorithm achieving a strong form of partial concurrency in bidirectional identified networks of arbitrary topology. As additional feature, this algorithm is *snap-stabilizing* [6]. *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* [14]: 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, 20, 22, 25]. In [5], Boulinier *et al.* propose a self-stabilizing unison algorithm which allows to solve local mutual exclusion, local group mutual exclusion, and local readers-writers problem. In [25], Nesterenko and Arora propose self-stabilizing algorithms for the solving the local mutual exclusion, dining philosophers, and drinking philosophers problems. There are also many self-stabilizing algorithms for local mutual exclusion [20, 22]. In [7], 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. Moreover, none of the aforementioned papers (especially [7]) consider the maximal concurrency issue. Finally, note

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

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 In the next section, we define 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 6. We prove its correctness in Section 7 and its partial concurrency in Section 8. We conclude in Section 9.

2 Computational Model and Specifications

2.1 Distributed Systems

We consider *distributed systems* made 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 connected 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 assume the *locally shared memory model* [14] 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 consisting in one state of each process. 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* [18]. 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* (see 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, *e.g.*, mutual exclusion, ℓ-exclusion, k-out-of-ℓ-exclusion, 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, depending on the considered problem. 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.

Note that the term "resource" is a used here in a broad sense. It actually encompasses various realities, depending on the context. In some applications, like in [21], the resource can be "virtual", while in some other the resource is a physical device. Furthermore, in some case, a resource actually represents a pool of resources of the same type. In this latter case, the pool of resources consists of multiple anonymous resources.

2.3.1 Local Resource Allocation

In the *Local Resource Allocation (LRA)* problem [7] each process requests at most one resource at a time. The problem 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. In contrast, there is no restriction for concurrently allocating the same resource to any number of processes that are not neighbors.

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 also means that any network considered from now on contains at least two processes.

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 [6], 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 requests for r in configuration γ_i. We assume that if $Request(\gamma_i, p, r)$ holds, it continuously holds until (at least) 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 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:

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

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 Maximal Concurrency

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 resources they are requesting without violating the safety of the considered resource allocation problem, then at least one of them should eventually satisfies its request, even if no process releases the resource(s) it holds meanwhile.

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

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

The first (resp. second) predicate means that no resource is released (resp. no new request occurs) between γ_i and γ_j. Notice that for any $i \geq 0$, $continuousCS(\gamma_i)$ and $noReq(\gamma_i)$ trivially hold.

Let $e = (\gamma_i)_{i \geq 0}$, $k \geq 0$ and $T \geq 0$. The function $R(e, k, T)$ is defined if and only if the execution $(\gamma_i)_{i \geq k}$ contains at least T rounds. In the case it is defined, the function returns $x \geq k$ such that the execution factor $\gamma_k \dots \gamma_x$ contains exactly T rounds.

Definition 1 (Maximal Concurrency)**.** A resource allocation algorithm $\mathcal{A}$ is *maximal concurrent* in a network $G = (V, E)$ if and only if

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

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 $continuousCS(\gamma\gamma') \wedge noReq(\gamma\gamma')$;

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

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

No Deadlock ensures that whenever a request can be satisfied, the algorithm has no deadlock and can still execute some step, even if no resource is released and no new request happens. **No Livelock** assumes that there exists a number of round N (which depends on the complexity of the algorithm, and henceforth on the network dimensions) such that: if during an execution, there exists some requests that can be satisfied, then at least one of them should be satisfied within N rounds, even if no resource is released and no new request happens meanwhile. Notice that the mention "no new request happens meanwhile" ensures that N uniquely depends on the algorithm and the network; if not, N would also depend on the scheduling of the requests.

3.2 Alternative Definition

We now provide an alternative definition of maximal concurrency: instead of constraining P_{Free} to decrease every N rounds during which there is neither new request, nor critical section exit, it expresses that P_{Free} becomes empty after enough rounds in such a situation.

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 new request occurs and no resource is released) during the execution factor $\gamma_i \dots \gamma_{endCS(e,i)}$ (resp. $\gamma_i \dots \gamma_{M(e,i)}$). Formally,

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

Note that $endCS(e, i)$ is always defined (for any e and any i) since $continuousCS(\gamma_i)$ holds and any critical section is assume to be finite. Consequently, $M(e, i)$ is always defined, since the set $\{j \geq i : noReq(\gamma_i \dots \gamma_j) \wedge j \leq endCS(e, i)\}$ is not empty and bounded by $endCS(e, i)$.

Definition 2 (Maximal Concurrency)**.** A resource allocation algorithm is *maximal concurrent* in a network $G = (V, E)$ if and only if

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 $continuousCS(\gamma\gamma') \wedge noReq(\gamma\gamma')$;

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

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

No Deadlock is identical in Definitions 1 and 2. However, **No Livelock** assumes now that there exists a (greater) number of rounds T_{MC} such that if no resource is released and no new request happens during T_{MC} rounds, then 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.

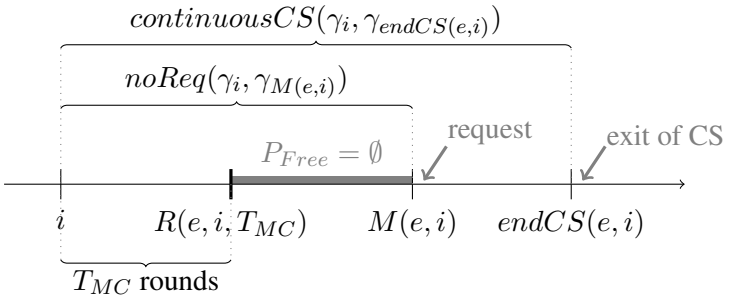


Figure 1: Illustration of Definition 2

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

Proof. Note first that the **No Deadlock** part is identical in both definitions. Consider now the **No Livelock** part: If Definition 1 holds, then Definition 2 holds by letting $T_{MC} = n \times N$; if Definition 2 holds, then Definition 1 holds by letting $N = T_{MC}$. □

Using Definition 2, remark that an algorithm is not maximal concurrent in a network $G = (V, E)$ if and only if

- 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 $continuousCS(\gamma\gamma') \wedge noReq(\gamma\gamma')$, there is no possible step of the algorithm from γ to γ' ($\gamma \not\mapsto \gamma'$);
- or **No Livelock** is violated: for every $T > 0$, there exists an execution $e = (\gamma_i)_{i \geq 0}$ and an index $i \geq 0$ such that $R(e, i, T)$ is defined, $R(e, i, T) \leq M(e, i)$, and $P_{Free}(\gamma_{R(e,i,T)}) \neq \emptyset$.

3.3 Instantiations

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 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 requested resource. Below, we denote by $\gamma(p).req$ the resource requested/used by process p in configuration γ. If p neither requests nor uses any resource, then $\gamma(p).req = \perp$, where $\perp$ is compatible with every resource. 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 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 The k-out-of-ℓ exclusion problem [10] 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, *i.e.*, $\gamma(p).req \in \{0, \ldots, k\}$. 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.* [10], 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 meantime.

In [10], 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 meantime. 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.

3.4 Strict (k, ℓ)-liveness versus Maximal Concurrency

As an illustrative example, we now show that the original definition of *strict (k, ℓ)-liveness* [10] is equivalent to the instantiation of maximal concurrency we propose in Example 3 of the previous subsection.

In [10], to introduce strict (k, ℓ)-liveness, the authors assume that a process can stay in critical section forever. Notice that this assumption is only used to define strict (k, ℓ)-liveness, critical sections are otherwise always assumed to be finite. Using this artifact, they express that a k-out-of-ℓ exclusion algorithm satisfies the *strict (k, ℓ)-liveness* in a network $G = (V, E)$ as follows: Let $I \subseteq V$ be the set of processes executing the critical section forever. Let $nbFree = \ell - \sum_{p \in I} p.req$. If there exists $p \in V$ such that p is requesting for $p.req \leq nbFree$ resources, then, eventually at least one requesting process (maybe p) enters the critical section.

Let $\mathcal{A}$ be a k-out-of-ℓ exclusion algorithm which is maximal concurrent in a network $G = (V, E)$. Assume an execution starting in configuration γ such that there is a set I of processes executing the critical section forever from γ. Assume also, for the purpose of contradiction, that from γ, no requesting process ever enters the critical section although there exists a requesting process p such that $p.req \leq nbFree$. Then, as the number of processes is finite, the system eventually reaches a configuration γ' from which no new request ever occur. By **No Deadlock**, the execution from γ' is infinite. Moreover, the daemon being weakly fair, every round from γ' is finite. Now, by **No Livelock**, after a finite number of rounds, one process enters the critical section (*n.b.*, P_{Free} is not empty because of p), a contradiction. Hence, $\mathcal{A}$ satisfies the strict (k, ℓ)-liveness in G.

Let $\mathcal{B}$ be a k-out-of-ℓ exclusion algorithm which is not maximal concurrent in a network $G = (V, E)$. Assume first $\mathcal{B}$ does not satisfy **No Deadlock**: there exists a configuration γ such that $P_{Free}(\gamma) \neq \emptyset$ and for every configuration γ' such that $continuousCS(\gamma\gamma') \wedge noReq(\gamma\gamma')$, there is no possible step of the algorithm from γ to γ'. Assume an execution from γ where all critical sections are infinite and there is no new request. Then, the system is deadlock and, consequently, no process of P_{Free} can enter the critical section. Now, P_{Free} is not empty. So, there exists a requesting process p such that $p.req \leq nbFree$. Moreover, only processes of P_{Free} can enter critical section without violating safety. Consequently, no

process ever enter the critical section during this execution: $\mathcal{B}$ does not satisfy the strict (k, ℓ)-liveness in G.

Finally, assume $\mathcal{B}$ violates **No Livelock**: for every $T > 0$, there exists an execution $e = (\gamma_i)_{i \geq 0}$ and an index $i \geq 0$ such that $R(e, i, T)$ is defined, $R(e, i, T) \leq M(e, i)$, and $P_{Free}(\gamma_{R(e,i,T)}) \neq \emptyset$. So, it is possible to build an infinite execution e, where all critical sections are infinite, no new request happens, and P_{Free} is never empty. As there is no new request, P_{Free} is never empty, and the number of processes is finite, there is an infinite suffix s of e where no process leaves P_{Free} (*i.e.*, no process of P_{Free} enters critical section) although P_{Free} is not empty. In s, there exists a requesting process p such that $p.req \leq nbFree$ because P_{Free} is not empty, but no process ever enter the critical section because only processes of P_{Free} can enter critical section without violating safety. Hence, $\mathcal{B}$ does not satisfy the strict (k, ℓ)-liveness in G.

Hence, the original definition of strict (k, ℓ)-liveness [10] is equivalent to the instantiation of maximal concurrency proposed in Example 3 of the previous subsection.

4 Maximal Concurrency versus Fairness

4.1 Necessary Condition on Concurrency in LRA

Maximal concurrency has been shown to be achievable in ℓ-exclusion [19]. However, there exist problems where it is not possible to ensure the maximal degree of concurrency, *e.g.*, Datta *et al.* showed in [10] that it is impossible to design a k-out-of-ℓ exclusion algorithm that satisfies the strict (k, ℓ)-liveness, which is equivalent to the maximal concurrency. Precisely, the impossibility proof shows that in this problem, fairness and maximal concurrency are incompatible properties. We now study the maximum degree of concurrency that can be achieved 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 is a technical result which will be used to show that there are (important) instances of the LRA problem for which it is impossible to design a maximal concurrent algorithm working in arbitrary networks (Theorem 1).

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

We recall that $\gamma(p).req$ denotes the resource requested/used by process p in configuration γ. If p neither requests nor uses any resource, then $\gamma(p).req = \perp$, where $\perp$ is compatible with every resource. We define the *conflicting neighborhood* of p in γ, denoted by $\mathcal{CN}_p(\gamma)$, as follows: $\mathcal{CN}_p(\gamma) = \{q \in \mathcal{N}_p : \gamma(p).req \not\perp \gamma(q).req\}$. Note that if p is not requesting, then $\mathcal{CN}_p(\gamma) = \emptyset$.

Below we consider any instance $\mathcal{I}$ of the LRA problem, where every process can request the same set of resources $\mathcal{R}$ (*i.e.*, $\forall p \in V, \mathcal{R}_p = \mathcal{R}$) and $\exists x \in \mathcal{R}$ such that $x \not\perp x$. Notice that the local mutual exclusion and the local readers-writers problem belong to this class of LRA problems.

Lemma 2. *For any algorithm solving $\mathcal{I}$ in a network $G = (V, E)$, if $|V| > 1$, then for any process p, there exists an execution $e = (\gamma_i)_{i \geq 0}$, with configuration γ_T, $T \geq 0$, and a process $q \in \mathcal{CN}_p(\gamma_T)$ 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 $e' = (\gamma'_i)_{i \geq 0}$ which shares the same prefix as e between γ_0 and γ_T (i.e., $\forall i \in \{0, ...T\}, \gamma_i = \gamma'_i$),

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

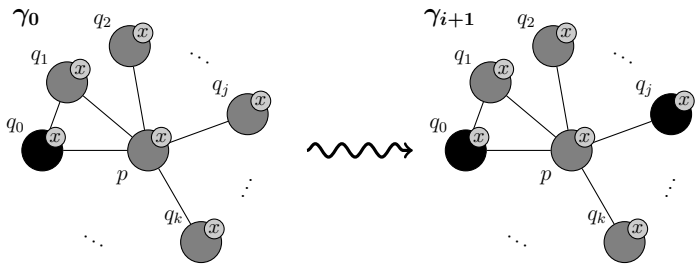


Figure 2: Outline of the execution $(\gamma_i)_{i\geq 0}$ of the proof of Lemma 2 on the neighborhood of p. Black nodes are in critical section, gray nodes are requesting.

Proof. Consider any algorithm solving $\mathcal{I}$ in a network $G = (V, E)$ with $|V| > 1$. Let $p \in V$.

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

Finally, 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 γ_0 in Figure 2.

Again, if $P_{Free}(\gamma_0) = \emptyset$, then we let $T = 0$ and Assertion 1 holds. Moreover, 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 q_0 holds x and no new request occurs, P_{Free} remains empty: Assertion 2 holds.

Assume now 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 (this is possible by the **No Deadlock** property). If no neighbor of p ever exits from P_{Free}, we are done Assertions 1 and 2 are both satisfied. Otherwise, let $i > 0$ and $j \in \{1, ..., k\}$ such that q_j is the first neighbor of p to exit from P_{Free} and $\gamma_i \mapsto \gamma_{i+1}$ is the first step where q_j exits from P_{Free}. We replace step $\gamma_i \mapsto \gamma_{i+1}$ by two steps $\gamma_i \mapsto \gamma'_{i+1} \mapsto \gamma'_{i+2}$:

- q_j leaves $P_{Free}(\gamma_i)$ and has access to x (by assumption) in $\gamma_i \mapsto \gamma'_{i+1}$,
- q_0 releases its critical section in $\gamma_i \mapsto \gamma'_{i+1}$ and requests again x in $\gamma'_{i+1} \mapsto \gamma'_{i+2}$.

Configuration γ'_{i+2} is shown in Figure 3. Hence, $P_{Req}(\gamma'_{i+2}) = \{p\} \cup \{q_l, l \neq j \wedge l \in \{0, ..., k\}\}$ and $P_{Free}(\gamma'_{i+2}) = \mathcal{CN}_p(\gamma'_{i+2}) \backslash \big(\{q_j\} \cup \mathcal{CN}_{q_j}(\gamma'_{i+2})\big) = \mathcal{N}_p \backslash (\{q_j\} \cup \mathcal{N}_{q_j})$. So, in γ'_{i+2} the system is in a situation similar to γ_0. If this scenario is repeated indefinitely, the algorithm never satisfies the request of p, contradicting the fairness of the LRA specification. 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 Assertion 1) and constant for some $q_l \in \mathcal{N}_p$, until q_l releases its resource or some new request occurs (this proves Assertion 2).

□

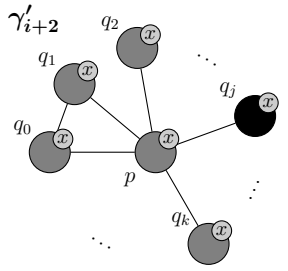


Figure 3: Neighborhood of p in configuration γ'_{i+2} in the proof of Lemma 2.

4.2 Impossibility Result

Theorem 1. *It is impossible to design a maximal concurrent algorithm solving $\mathcal{I}$ in every network.*

Proof. Assume, by the contradiction, that there is a maximal concurrent algorithm solving $\mathcal{I}$ in every network. Consider a network (of at least two processes) which contains a process p, such that $\forall q \in \mathcal{N}_p$, $\mathcal{N}_p \backslash (\{q\} \cup \mathcal{N}_q) \neq \emptyset$. (Take for instance a star network where p is at the center.)

From Lemma 2, there exists $e = (\gamma_i)_{i \geq 0}$ 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 $e' = (\gamma'_i)_{i \geq 0}$ 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 the **No Livelock** property of 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 $P_{Free}(\gamma_T) \neq \emptyset$. By applying the second part of Lemma 2, we have $P_{Free}(\gamma'_{T+1}) = P_{Free}(\gamma_T) \neq \emptyset$. 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, $P_{Free}(\gamma'_{R(e',T,T_{MC})}) = P_{Free}(\gamma_T) \neq \emptyset$, contradicting the **No Livelock** property of the maximal concurrency.

□

5 Partial Concurrency

We now generalize the maximal concurrency to be able to define weaker degree of concurrency that will be achievable for all instances of LRA. This generalization is called *partial concurrency*.

5.1 Definition

Maximal concurrency requires that a requesting process should not be prevented from accessing its critical section unless to avoid safety violations. The idea of partial concurrency is to slightly relax this property by (momentarily) blocking some requesting processes that nevertheless could enter their critical section without violating safety. We define $\mathcal{P}$ as a predicate which represents the sets of requesting processes that can be (momentarily) blocked, while they could access their requesting resources without violating safety.

Definition 4 (Partial Concurrency *w.r.t.* $\mathcal{P}$)**.** A resource allocation algorithm $\mathcal{A}$ is *partially concurrent* w.r.t. $\mathcal{P}$ in a network $G = (V, E)$ if and only if

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

No Livelock: There exists a number of rounds T_{PC} such that for every execution $e = (\gamma_i)_{i \geq 0}$ 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 maximal concurrency is equivalent to partial concurrency *w.r.t.* $\mathcal{P}_{max}$, where $\forall X \subseteq V, \forall \gamma \in \mathcal{C}, \mathcal{P}_{max}(X, \gamma) \equiv X = \emptyset$.

5.2 Strong Concurrency

The proof of Lemma 2 exhibits a possible scenario for some instances of LRA which shows the incompatibility of fairness and maximal concurrency: enforce maximal concurrency can lead to unfair behaviors where some neighbors of a process alternatively use resources which are conflicting with its own request. So, to achieve fairness, we must then relax the expected level of concurrency in such a way that this situation cannot occur indefinitely. The key idea is that sometimes the algorithm should prioritize one process p against its neighbors, although it cannot immediately enter the critical section because some of its conflicting neighbors are in critical section. In this case, the algorithm should momentarily block all conflicting requesting neighbors of p that can enter critical section without violating safety, so that p enters critical section first. In the worst case, p has only one conflicting neighbor q in critical section and so the set of processes that p has to block contains up to all conflicting (requesting) neighbors of p that are neither q, nor conflicting neighbors of q (by definition, any conflicting neighbor common to p and q cannot access critical section without violating safety because of q). We derive the following refinement of partial concurrency based on this latter observation. This property seems to be very close to the maximum degree of concurrency which can be ensured by an algorithm solving all instances of LRA.

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

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

In the next section, we propose a strongly concurrent LRA algorithm.

6 Local Resource Allocation Algorithm

We now propose a snap-stabilizing LRA algorithm which achieves strong concurrency.

6.1 Overview of the solution

The overall idea of our algorithm is the following. To maximize concurrency, our algorithm should follow, as much as possible, a greedy approach: if there are requesting processes having no conflicting neighbor in the critical section, then those which have locally the highest identifier are allowed to enter critical section.

Now, the algorithm should not be completely greedy, otherwise livelock can occur at processes with low identifiers, violating the fairness of the specification.

So, the idea is to make circulating a token whose aim is to cancel the greedy approach, but only in the neighborhood of the tokenholder (the rest of the network continue to follow the greedy approach): the

tokenholder, if requesting, has the priority to satisfy its request; all its conflicting neighbors are blocked until it accesses its critical section. To ensure fairness, these blockings take place even if the tokenholder cannot currently access its critical section (because maybe one of its conflicting neighbor is in critical section). Such blockings slightly degrade the concurrency, this is why our algorithm is strong, but not maximal, concurrent.

6.2 Composition

Composition techniques are important in the self-stabilizing area because they allow to simplify the design, analysis, and proofs of algorithms. Consider an arbitrary composition operator $\oplus$, and two algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$. Let e be an execution of $\mathcal{A}_1 \oplus \mathcal{A}_2$. Let $i \in \{1, 2\}$. We say that e is *weakly fair w.r.t.* $\mathcal{A}_i$ if there is no infinite suffix of e in which a process does not execute any action of $\mathcal{A}_i$ while being continuously enabled *w.r.t.* $\mathcal{A}_i$.

Our algorithm consists of the composition 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. These two modules are composed using a *fair composition* [16], denoted by $\mathcal{LRA} \circ \mathcal{TC}$. In such a composition, each process executes a step of each algorithm alternately. Recall that the purpose of this composition is, in particular, 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 $\mathcal{X}$ and $\mathcal{Y}$. The equivalent non-composite algorithm $\mathcal{Z}$ can be obtained by applying the following rewriting rule: In $\mathcal{Z}$, a process has its variables in $\mathcal{X}$, those in $\mathcal{Y}$, and an additional variable $b \in \{1, 2\}$. Assume now that $\mathcal{X}$ is composed of x actions denoted by

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

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

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

Then, $\mathcal{Z}$ is composed of the following $x + y + 2$ actions:

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

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

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

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

Notice that, by definition of the composition, under the weak fair daemon assumption, no algorithm in the composition can prevent the other from executing, if this latter is continuously enabled. Rather, it can only slow down the execution by a factor 2.

Remark 1. Under the weakly fair daemon, in $\mathcal{A}_1 \circ \mathcal{A}_2$ we have: $\forall i \in \{1, 2\}, \forall p \in V$, if p is continuously enabled *w.r.t.* $\mathcal{A}_i$ until (at least) executing an enabled action of $\mathcal{A}_i$, then p executes an enabled action of $\mathcal{A}_i$ within at most 2 rounds.

Remark 2. Under the weakly fair daemon, $\forall i \in \{1, 2\}$, every execution of $\mathcal{A}_1 \circ \mathcal{A}_2$ is *weakly fair w.r.t.* $\mathcal{A}_i$.

6.3 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 $PassToken(p)$[3]. 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)**.** *Consider an arbitrary composition of $\mathcal{TC}$ and some other algorithm. Let e be any execution of this composition which is* weakly fair w.r.t. $\mathcal{TC}$*.*

If for any process p, $PassToken(p)$ is executed in e only when $TokenReady(p)$ holds, then $\mathcal{TC}$ stabilizes in e, 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. *Consider an arbitrary composition of $\mathcal{TC}$ and some other algorithm. Let e be any execution of this composition which is* weakly fair w.r.t. $\mathcal{TC}$ *and where $\mathcal{TC}$ is stabilized.*

Then, $\forall p \in V$, each time $TokenReady(p)$ holds in e, $TokenReady(p)$ is continuously true in e until $PassToken(p)$ is invoked.

Property 3 (Fairness)**.** *Consider an arbitrary composition of $\mathcal{TC}$ and some other algorithm. Let e be any execution of this composition which is* weakly fair w.r.t. $\mathcal{TC}$ *and where $\mathcal{TC}$ is stabilized.*

If $\forall p \in V$,

- *$PassToken(p)$ is invoked in e only when $TokenReady(p)$ holds, and*
- *$PassToken(p)$ is invoked within finite time in e each time $TokenReady(p)$ holds,*

then $\forall p \in V$, $TokenReady(p)$ holds infinitely often in e.

To design $\mathcal{TC}$, we proceed as follows. There exist several self-stabilizing token circulations for arbitrary rooted networks [8, 11, 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, 17, 12, 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 Properties 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.

[3] Since $\mathcal{TC}$ is a black box with only two outputs: $TokenReady(p)$ and $PassToken(p)$, these outputs are the only part of $\mathcal{TC}$ that $\mathcal{LRA}$ can use.

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 $\mathcal{TC}$, 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

$$
\begin{array}{lcl}
ResourceFree(p) & \equiv & \forall q \in \mathcal{N}_p, \big(q.status = \mathsf{In} \Rightarrow p.req \rightleftharpoons q.req\big) \\
IsBlocked(p) & \equiv & \neg ResourceFree(p) \vee \big(\exists q \in \mathcal{N}_p, q.status = \mathsf{Blocked} \\
 & & \quad \wedge q.token \wedge p.req \neq q.req\big)
\end{array}
$$

Guards

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

Actions

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

6.4 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 (page 5) 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 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 $p.req = \bot$, 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 the application requests. 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 be switched from $\bot$ to $r \in \mathcal{R}_p$ if and only if $p.status = \mathsf{Out}$,*
- *The value of $p.req$ can be switched from $r \in \mathcal{R}_p$ to $\bot$ (meaning that the application leaves the critical section) if and only if $p.status = \mathsf{In}$.*
- *The value of $p.req$ cannot be directly switched 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 $\bot$.*

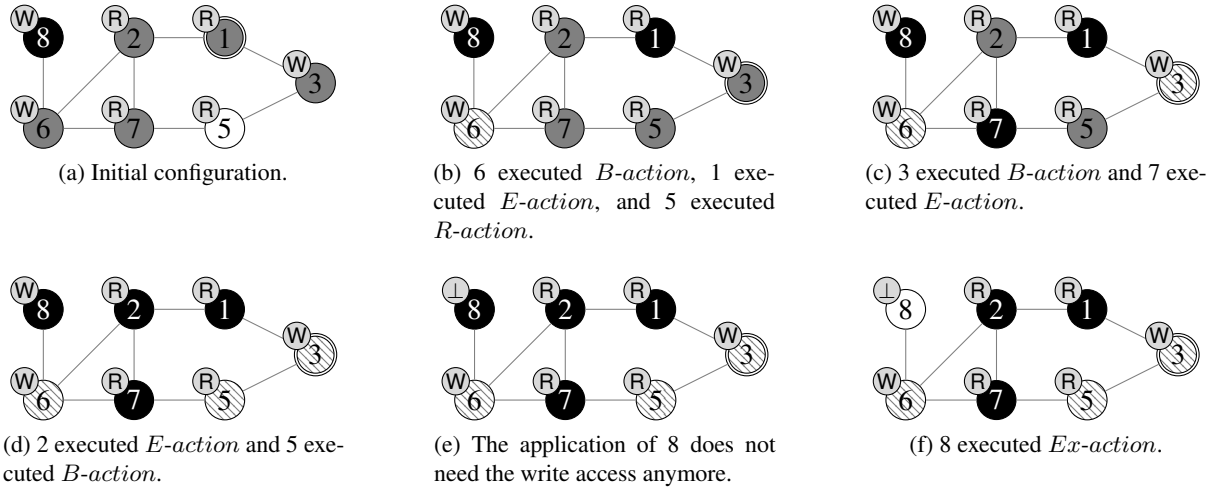
Consequently, the predicate $Request(\gamma_i, p, r)$ in Specification 1 is given by $Request(\gamma_i, p, r) \equiv \gamma_i(p).req = r$.

The predicate $NoConflict(\gamma_i, p)$ is expressed by $NoConflict(\gamma_i, p) \equiv \gamma_i(p).status = \mathsf{In} \Rightarrow \big(\forall q \in \mathcal{N}_p, \gamma_i(q).status = \mathsf{In} \Rightarrow (\gamma_i(q).req \rightleftharpoons \gamma_i(p).req)\big)$. (Remind that $\bot$ 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: $Start(\gamma_i, \gamma_{i+1}, p, r) \equiv \gamma_i(p).status = \mathsf{Out} \ \wedge \ \gamma_{i+1}(p).status = \mathsf{Wait} \ \wedge \ \gamma_i(p).req = \gamma_{i+1}(p).req = r$.

A computation $\gamma_i \dots \gamma_j$ where $Result(\gamma_i \dots \gamma_j, p, r)$ holds means that 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 $\gamma_j \mapsto \gamma_{j+1}$. So, $Result(\gamma_i \dots \gamma_j, p, r) \equiv \gamma_i(p).status = \mathsf{Wait} \ \wedge \ \gamma_i(p).req = \gamma_{i+1}(p).req = r \ \wedge \ \forall k \in \{i+1 \dots j-1\}, \gamma_k(p).status = \mathsf{In} \ \wedge \ \gamma_j(p).status = \mathsf{Out} \ \wedge \ \gamma_j(p).req = \bot$.

We now illustrate the principles of $\mathcal{LRA}$ with the example given in Figure 4. In this example, we consider the local readers-writers 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 hatched 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.



(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 $\mathcal{LRA} \circ \mathcal{TC}$.

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 $R\text{-}action$: 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 $B\text{-}action$ (see 6 in (a)↦(b)). If there is no more neighbor using conflicting resources, p gets back to status Wait by $UB\text{-}action$.

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 $\big(p.token \wedge \neg q.token\big) \vee \big(p.token = q.token \wedge p > q\big)$[4]. More precisely, if there is no waiting tokenholder 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 tokenholders (there may be several tokens during the stabilization phase of $\mathcal{TC}$) block all their requesting neighbors, except the ones requesting for non-conflicting resources until they 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 $E\text{-}action$, *e.g.*, 7 in step (b)↦(c) or 1 in (a)↦(b). Moreover, if it holds a token, a tokenholder releases it when accessing its requested resource. Notice that, as a process is not blocked when one of its neighbors is requesting/using a compatible resource, several neighbors requesting/using compatible resources can concurrently enter/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 $\perp$. Then, p executes $Ex\text{-}action$ and switches its status to Out, *e.g.*, 8 during step (e)↦(f).

$RlT\text{-}action$ 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 no resource requested 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, $RsT\text{-}action$, ensures the consistency of variable $token$ so that the neighbors realize whether or not a process holds a token.

Hence, any request is satisfied in a finite time. As an illustrative example, consider the local mutual exclusion problem and the execution given in Figure 5. In this example, we try to delay as much as

[4]Notice that when two neighbors simultaneously hold the token (only during the stabilization phase of $\mathcal{TC}$), the one with the highest identifier has priority.

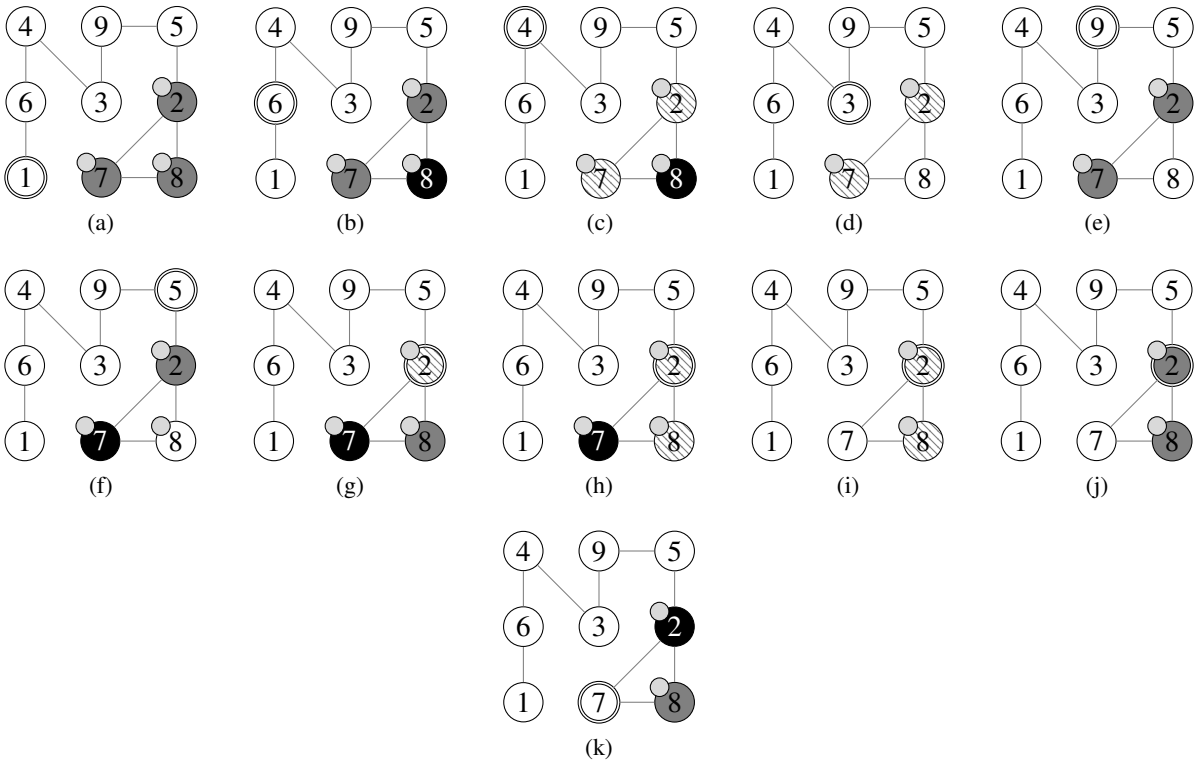


Figure 5: Example of execution of $\mathcal{LRA} \circ \mathcal{TC}$ on the local mutual exclusion problem. The bubbles mark the requesting processes.

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).

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

7.1 Correctness

In this subsection, we prove that $\mathcal{LRA} \circ \mathcal{TC}$ is snap-stabilizing *w.r.t.* SP_{LRA} (see Specification 1, page 5), assuming a distributed weakly fair daemon. First, we show the safety part, namely, the Resource Conflict Freedom property is always satisfied. Then, we assume a distributed weakly fair daemon to prove the liveness part, *i.e.*, the 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. Since by definition, $p, q \in Candidates(p)$ and $p, q \in Candidates(q)$, we obtain 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 be switched from $\perp$ in γ to $r \in \mathcal{R}_p$ in γ' only if $\gamma(p).status = \mathsf{Out}$. But $\gamma'(p).status = \mathsf{In}$ and it is impossible to switch $p.status$ from Out to In in one step.
- The value of $p.req$ cannot be switched 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 $\gamma(q).req \neq \gamma(p).req$. 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 $\gamma(q).req \neq \gamma(p).req$. 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 although 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. By definition of the algorithm, for any process p, $PassToken(p)$ is executed only in $\mathcal{LRA}$ when $TokenReady(p)$ holds (see RlT-action and E-action). Moreover, by Remark 2, every execution of $\mathcal{LRA} \circ \mathcal{TC}$ is *weakly fair w.r.t.* $\mathcal{TC}$. So, $\mathcal{TC}$ self-stabilizes to a unique tokenholder in every execution of $\mathcal{LRA} \circ \mathcal{TC}$, by Property 1. □

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

Proof. Let e be an 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, the values of $token$ variables are eventually updated to the corresponding value of the predicate $TokenReady$. Indeed, the values of predicate $TokenReady$ do not change anymore. So, if there is $x \in V$ such that $x.token \neq TokenReady(x)$, $RsT\text{-}action$ (the highest priority action of $\mathcal{LRA}$) is continuously enabled at x, until x executes it. Now, by Remark 1, in finite time, x executes $RsT\text{-}action$ to update 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 continuously disabled from γ'. Then, we can distinguish six cases:

1. If $\gamma'(p).status = \mathsf{Wait}$ and $\gamma'(p).req \neq \bot$, then $TokenCand(p) = \{p\}$ 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. $E\text{-}action$ is disabled forever at q from 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 $\bot$ is compatible with any resource, $\gamma'(q).req \neq \bot$. Using Property 5, in finite time the request of q becomes $\bot$ and remains $\bot$ until q obtains $status$ Out (Property 4). So $Ex\text{-}action$ is continuously enabled at q, until q executes it. Hence, by Remark 1, 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 $\mathsf{Blocked}$ in the meantime, $UB\text{-}action$ is continuously enabled at p until p executes it, so p gets back status Wait in finite time by Remark 1. Then, $Winner(p) = p$ still holds so $E\text{-}action$ is continuously enabled at p, until p executes it. Hence, by Remark 1, in finite time, p executes $E\text{-}action$ and releases its token, a contradiction.

2. If $\gamma'(p).status = \mathsf{Out}$ and $\gamma'(p).req \neq \bot$, the application cannot modify $p.req$ 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 until p executes it, and p eventually gets status Wait by Remark 1. We then reach case 1 and we are done.

3. If $\gamma'(p).status = \mathsf{Out}$ and $\gamma'(p).req = \bot$. If eventually $p.req \neq \bot$, then we retrieve case 2, a contradiction. Otherwise, $RlT\text{-}action$ is continuously enabled at p until p executes it. So, by Remark 1, in finite time, p executes $RlT\text{-}action$ and releases its token by a call to $PassToken(p)$, a contradiction.

4. If $\gamma'(p).status = \mathsf{Blocked}$ and $\gamma'(p).req \neq \bot$, then $p.status = \mathsf{Blocked}$ forever from γ', otherwise we eventually retrieve case 1. So, $\forall q \in \mathcal{N}_p$ 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 $\gamma'(p).req \neq \gamma'(q).req$, we have $q.status \neq \mathsf{In}$ forever after a finite time. So, eventually $UB\text{-}action$ is continuously enabled at p until p executes it. Hence, by Remark 1, in finite time, p gets $status$ Wait and we retrieve case 1, a contradiction.

5. If $\gamma'(p).status \in \{\mathsf{Wait}, \mathsf{Blocked}\}$ and $\gamma'(p).req = \bot$. If eventually $p.req \neq \bot$, then we retrieve cases 1 or 4, a contradiction. Otherwise, $Ex\text{-}action$ is continuously enabled at p until p executes it. So, by Remark 1, in finite time, p executes $Ex\text{-}action$ and we retrieve case 3, a contradiction.

6. If $\gamma'(p).status = \mathsf{In}$, either $\gamma'(p).req = \bot$ or in finite time $p.req$ becomes $\bot$ (Property 5) and remains $\bot$ until p obtains $status$ Out (Property 4). Once $p.req = \bot$, $Ex\text{-}action$ is continuously enabled at p until p executes it. So, by Remark 1 p eventually gets status Out, and we retrieve case 3, a contradiction.

□

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. *If $Exit(p)$ continuously holds at some process p until it executes $Ex\text{-}action$, then p executes $Ex\text{-}action$ in finite time.*

Proof. Assume, by the contradiction, that from some configuration $Exit(p)$ continuously holds, but p never executes $Ex\text{-}action$. Then, remind that $RlT\text{-}action$ and $E\text{-}action$ are the only actions allowing p to release a token. Now, $Exit(p)$ is the guard of action $Ex\text{-}action$ whose priority is higher than those of $RlT\text{-}action$ and $E\text{-}action$. So, p never more releases a token, a contradiction to Lemma 5 and Corollary 1. □

Lemma 8. *If $Requested(p)$ continuously holds at some process p until it executes $R\text{-}action$, then p executes $R\text{-}action$ in finite time.*

Proof. Assume, by the contradiction, that from some configuration $Requested(p)$ continuously holds, but p never executes $R\text{-}action$. Then, remind that $RlT\text{-}action$ and $E\text{-}action$ are the only actions allowing p to release a token. Now, $Requested(p)$ implies $\neg ReleaseToken(p)$, so $RlT\text{-}action$ is disabled at p forever. Moreover, $Requested(p)$ is the guard of action $R\text{-}action$ whose priority is higher than the one of $E\text{-}action$. So, p never more releases a token, a contradiction to Lemma 5 and Corollary 1. □

Lemma 9. *Any process p such that $p.status \in \{\mathsf{Wait}, \mathsf{Blocked}\}$ and $p.req \neq \bot$ executes $E\text{-}action$ in finite time.*

Proof. Let e be an execution, $\gamma \in e$ be a configuration, and $p \in V$ such that $\gamma(p).status \in \{\mathsf{Wait}, \mathsf{Blocked}\}$ and $\gamma(p).req \neq \bot$. Then, $p.req \neq \bot$ holds while $p.status \neq \mathsf{In}$ (Property 4). So, while p does not execute $E\text{-}action$, $p.status \in \{\mathsf{Wait}, \mathsf{Blocked}\}$ and $p.req \neq \bot$. Now, by Lemma 5, the token circulation eventually stabilizes. By Corollary 1, in finite time p holds the unique token. From this configuration, p cannot keep forever the token (Lemma 6) and p can only release it by executing $E\text{-}action$ (by Property 2). □

Lemma 10. *Any process of $status$ different from* Out *sets its variable $status$ to* Out *within finite time.*

Proof. Let $p \in V$. Let γ be a configuration. Assume first $\gamma(p).status = \mathsf{In}$. If $\gamma(p).req \neq \bot$, in finite time $p.req$ is set to $\bot$ (Property 5) and then cannot be modified until p gets $status$ Out (Property 4). So, $Exit(p)$ continuously holds until p executes $Ex\text{-}action$. Then, $Ex\text{-}action$ is executed by p in finite time, by Lemma 7: p gets $status$ Out.

Assume now that $\gamma(p).status \in \{\mathsf{Wait}, \mathsf{Blocked}\}$. If eventually $p.req \neq \bot$, then p executes $E\text{-}action$ in a finite time (Lemma 9). So, p eventually gets $status$ In and we retrieve the previous case. Otherwise, $Exit(p)$ continuously holds until p executes $Ex\text{-}action$. Then, $Ex\text{-}action$ is executed by p in finite time, by Lemma 7: p 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, by Lemma 10, let say in $\gamma_{j-1} \mapsto \gamma_j$ ($j \geq k$). Now, if $\gamma_j(p).req \neq \bot$ holds, it holds continuously while $p.status =$ Out (Property 4). So, $Requested(p)$ continuously holds until p executes $R\text{-}action$. By Lemma 8, p eventually executes $R\text{-}action$, let say in $\gamma_l \mapsto \gamma_{l+1}$, $l \geq j \geq k$. Then, $\gamma_{l+1}(p).status =$ Wait. Notice that the application of p cannot modify 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 $\gamma_{k+1}(p).status =$ Wait and $\gamma_{k+1}(p).req = r$. Using Lemma 9, 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 modify the value of req until p obtains $status$ In (Property 4) so $\gamma_{l-1}(p).req = \gamma_l(p).req = \gamma_{k+1}(p).req = r$. By Property 5 and from the algorithm, $p.status =$ In while $p.req \neq \bot$ and the application sets within finite time $p.req$ to $\bot$ (this is the only modification that can be made on $p.req$). Then, $p.req = \bot$ until $p.status =$ Out, still by Property 5, and from the algorithm, p can only switch $p.status$ from In to Out. So, $p.req = \bot$ continuously and $Exit(p)$ continuously holds until p executes $Ex\text{-}action$. Then, by Lemma 7: there is a step $\gamma_{l'} \mapsto \gamma_{l'+1}$ (with $l' \geq l$), where p executes $Ex\text{-}action$ to switch $p.status$ from In to Out. So, $\gamma_{l'}(p).status =$ In and $\gamma_{l'+1}(p).status =$ Out. Consequently, $Result(\gamma_l \dots \gamma_{l'}, p, r)$ 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 this subsection, we analyze the waiting time, *i.e.*, the number of rounds required to obtain critical section after a request. Here, we assume that the execution of critical section *lasts at most C rounds*.

Lemma 11. *In $\mathcal{LRA} \circ \mathcal{TC}$, after stabilization of $\mathcal{TC}$, there are at most $2C + 14$ rounds between a step where $TokenReady(p)$ becomes true and the execution of $PassToken(p)$.*

Proof. As $PassToken$ is only executed in the $\mathcal{LRA}$ part of $\mathcal{LRA} \circ \mathcal{TC}$, we focus on counting rounds from $\mathcal{LRA}$, first. Then, the result has to be multiply by 2 due to the composition with $\mathcal{TC}$ (Remark 1).

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

Assume $TokenReady(p)$ holds. In one round, the variables $token$ are correctly evaluated thanks to $RsT\text{-}action$ executions (remind that $RsT\text{-}action$ is the highest priority action). Then, there are three cases:

1. Assume p is requesting but does not get the critical section yet. In the worst case, $p.status =$ Out and $p.req \neq \bot$. 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, they end their critical section (*i.e.*, their variable req becomes $\bot$) within the C next rounds and p executes $B\text{-}action$ during the first of these C rounds. Notice that, as p holds the unique token and the $token$ variables are correctly evaluated, no other neighbor of p can enter the critical section meanwhile. In the worst case, every neighbor is out of the critical section (*i.e.*, their variable req becomes $\bot$, which is compatible with any other resource) after these C rounds. Finally, p is no more blocked and executes $UB\text{-}action$ in one round before executing $E\text{-}action$ within another round. Executing $E\text{-}action$, p releases its token. Hence, overall p releases its token within $C + 4$ rounds in this case.

2. Assume $p.req = \perp$. If $p.req$ becomes different from $\perp$ within one round, then this means that $p.status = \mathsf{Out}$ (Property 4) and we retrieve the previous case and overall p releases its token within $C + 5$ rounds. Otherwise, p satisfies $p.status = \mathsf{Out}$ in one round, by $Ex\text{-}action$ if necessary. Again, either $p.req$ becomes different from $\perp$ within the next round, we retrieve the previous case, and overall p releases its token within $C + 6$ rounds, or p executes $RlT\text{-}action$ during the round. So, in this latter case, p releases its token within 3 rounds.

3. Assume $p.status = \mathsf{In}$ and $p.req \neq \perp$. If $p.req$ becomes $\perp$ within one round, we retrieve the case 2. So overall p releases its token within $C+7$ rounds. Otherwise, $RlT\text{-}action$ is continuously enabled and executed by p within the round. So, p releases its token within two rounds in this latter case.

□

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 $T : TokenReady(p) \rightarrow PassToken(p)$ to $\mathcal{TC}$, see Remark 3, page 15) after its stabilization. Let N_{tok} be a bound on the number of $PassToken$ realized between two consecutive executions of $PassToken$ at the same process.

Theorem 6 (Waiting Time)**.** *A requesting process obtains access to critical section in at most* $2(T_s + T_{tok}) + (2C + 14) \times (N_{tok} + 1)$ *rounds.*

Proof. Let $p \in V$ such that $p.req \neq \perp$ and $p.status \neq \mathsf{In}$. In the worst case, p must wait to hold a token and be the unique tokenholder to get its critical section. $\mathcal{TC}$ stabilizes in $2T_S$ rounds (the factor 2 comes from the composition, see Remark 1). Then, in at most $2T_{tok} + (2C + 14) \times N_{tok}$, p gets the token, since it has to wait $2T_{tok}$ rounds due to Algorithm $\mathcal{TC}$ (again, the factor 2 comes from the composition, see Remark 1) and $(2C + 14) \times N_{tok}$ rounds due to Algorithm $\mathcal{LRA}$. Indeed, while executing action $T : TokenReady \rightarrow PassToken$ is atomic in $\mathcal{TC}$*, a process keeps the token at most $2C + 14$ rounds in $\mathcal{LRA} \circ \mathcal{TC}$ (Lemma 11). Finally, to obtain critical section, it is required that p executes $E\text{-}action$ which also releases the token: by Lemma 11 again, this may require $2C + 14$ additional rounds. Hence, in at most $2(T_s + T_{tok}) + (2C + 14) \times (N_{tok} + 1)$ rounds, p obtains its critical section.

□

For example, if we choose to build $\mathcal{TC}$ from the leader election algorithm given in [1] and the token circulation algorithm for arbitrary rooted networks introduced by Cournier *et al.* in [8], then T_s and T_{tok} are in $O(n)$ rounds, while N_{tok} is in $O(n)$ executions of $PassToken$. Applying these results to Theorem 6 shows that the waiting time is achievable in $O(C \times n)$ rounds. Notice also that this implementation of $\mathcal{TC}$ has a memory requirement of $\Theta(\log n)$ bits per process. Hence, $\mathcal{LRA} \circ \mathcal{TC}$ can be implemented using $\Theta(\log n + \log |\mathcal{R}_p|)$ per process p.

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

We first prove **No Deadlock**.

Lemma 12. *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 $P_{Free}(\gamma) \not\subseteq X$, there exists a configuration γ' and a step $\gamma \mapsto \gamma'$ such that $continuousCS(\gamma \ldots \gamma') \wedge noReq(\gamma \ldots \gamma')$.*

Proof. (By the contrapositive.) Assume a configuration γ where no action of the algorithm is enabled. First, if $P_{Free}(\gamma) = \emptyset$, we are done. So from now on, assume $P_{Free}(\gamma) \neq \emptyset$. In γ, $\mathcal{TC}$ has stabilized, by Lemma 5. So,

Claim 1. There is a unique tokenholder, t, in γ.
Moreover, as $RsT\text{-}action$ is disabled at every process, Claim 1 implies:

Claim 2. For every process p, $\gamma(p).token$ if and only if $p = t$.

Claim 3. $\gamma(t).status \neq \mathsf{Wait}$.
Proof of the claim: If $\gamma(t).status = \mathsf{Wait}$, then since $t = Winner(\gamma(t))$ (by Claim 2), either $\neg IsBlocked(\gamma(t))$ holds and $E\text{-}action$ is enabled at t, or $B\text{-}action$ is enabled at t, a contradiction.

Claim 4. $\forall p \in P_{Free}(\gamma)$, $\gamma(p).status = \mathsf{Blocked}$.
Proof of the claim: First, by definition, $\gamma(p).status \in \{\mathsf{Wait}, \mathsf{Blocked}, \mathsf{Out}\}$ and $\gamma(p).req \neq \bot$. If $\gamma(p).status = \mathsf{Out}$, $R\text{-}action$ is enabled at p, a contradiction. If $\gamma(p).status = \mathsf{Wait}$, then, $\neg IsBlocked(\gamma(p))$ since otherwise $B\text{-}action$ is enabled at p in γ. Consequently, $p \neq Winner(\gamma(p))$ holds, otherwise $E\text{-}action$ is enabled at p. So, we can build a sequence of processes $r_0, r_1, \ldots, r_k$ where $r_0 = p$ and such that $\forall i \in \{1, \ldots, k\}$, $r_i = Winner(r_{i-1})$. (Notice that none of the r_i are the tokenholder, since the tokenholder does not have status Wait, by Claims 1 and 3.) This sequence is finite because $r_0 < r_1 < \cdots < r_k$ (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 to execute $E\text{-}action$, a contradiction. Hence, $\gamma(p).status = \mathsf{Blocked}$.

Claim 5. $\forall p \in P_{Free}(\gamma)$, $p \in \mathcal{CN}_t(\gamma)$ and $\gamma(t).status = \mathsf{Blocked}$.
Proof of the claim: By Claim 4 and the fact that $UB\text{-}action$ is disabled at every process, we have $IsBlocked(\gamma(p))$ for every process $p \in P_{Free}(\gamma)$. Then, $p \in P_{Free}(\gamma)$ implies $ResourceFree(p)$ in γ, so by Claims 1 and 2, we can conclude.

Claim 6. There exists a neighbor q of t whose $status$ is In in γ.
Proof of the claim: By Claim 5, $\gamma(t).status = \mathsf{Blocked}$, so $IsBlocked(\gamma(t))$ since $UB\text{-}action$ is disabled at t. Now, by Claim 2, $IsBlocked(\gamma(t))$ implies that $\neg ResourceFree(t)$ holds in γ, which proves the claim.

By definition, a consequence of Claim 6 is that q and every process $p \in \mathcal{CN}_q(\gamma)$ do not belong to $P_{Free}(\gamma)$. Hence, Claims 5 and 6 imply that $\forall p \in P_{Free}(\gamma)$, $p \in \mathcal{CN}_t(\gamma) \setminus (\{q\} \cup \mathcal{CN}_q(\gamma))$. So, by letting $X = \mathcal{CN}_t(\gamma) \setminus (\{q\} \cup \mathcal{CN}_q(\gamma))$, we have $\mathcal{P}_{strong}(X, \gamma)$ and $P_{Free}(\gamma) \subseteq X$, and we are done. □

We now prove **No Livelock**.

Lemma 13. *Let $e = (\gamma_j)_{j \geq 0}$ be an execution of $\mathcal{LRA} \circ \mathcal{TC}$, $i \geq 0$ such that $\mathcal{TC}$ is stabilized in γ_i (i is defined by Lemma 5), and $t \in V$ the unique tokenholder in γ_i. If $R(e, i, 6)$ exists, $R(e, i, 6) \leq M(e, i)$, and $\forall j \in \{i + 1, \ldots, R(e, i, 6)\}$, $PassToken(t)$ is not executed in step $\gamma_{j-1} \mapsto \gamma_j$, then for every $k \in \{R(e, i, 4), \ldots, M(e, i)\}$:*

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

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

Claim 1. $\forall j \in \{R(e, i, 2), \ldots, R(e, i, 6)\}$, $\forall p \in V$, $\gamma_j(p).token = \texttt{true}$ if and only if $p = t$.
Proof of the claim: By hypothesis, $TokenReady(p)$ is constant between γ_i and $\gamma_{R(e,i,6)}$. So, if $p.token = TokenReady(p)$ in some configuration γ between γ_i and $\gamma_{R(e,i,2)}$, then $p.token = TokenReady(p)$ holds in all configurations between γ and $\gamma_{R(e,i,6)}$, since $RsT\text{-}action$ is disabled at p in all these configurations. Since by hypothesis, $TokenReady(p) \equiv (p = t)$ in all configurations between γ_i and $\gamma_{R(e,i,6)}$,

we are done. Assume, otherwise, that $p.token \neq TokenReady(p)$ in all configurations between γ_i and $\gamma_{R(e,i,2)}$, then $RsT\text{-}action$ (the highest priority action) is continuously enabled at p until p executes it. Now, in this case, p executes it within at most 2 rounds (Remark 1), hence, there is a configuration between γ_i and $\gamma_{R(e,i,2)}$, where $p.token = TokenReady(p)$, a contradiction.

Claim 2. $\gamma_{R(e,i,4)}(t).status = \mathsf{Blocked}$.
Proof of the claim: Assume, by the contradiction, that $\gamma_{R(e,i,4)}(t).status \neq \mathsf{Blocked}$.

(a) Assume $\gamma_{R(e,i,2)}(t).status = \mathsf{In}$. If $t.req = \perp$, then $t.req = \perp$ holds in all configurations between γ_i and $\gamma_{R(e,i,6)}$, by hypothesis. Moreover, $RsT\text{-}action$ is disabled at t in all configurations between $\gamma_{R(e,i,2)}$ and $\gamma_{R(e,i,6)}$, by Claim 1. Hence, by Remark 1, t executes $Ex\text{-}action$ within two rounds from $\gamma_{R(e,i,2)}$, and then $RlT\text{-}action$ within at most two more rounds. By this latter action, t releases the token by $PassToken(t)$, a contradiction.

Assume now that $t.req \neq \perp$ in $\gamma_{R(e,i,2)}$. Then, by hypothesis, $t.req \neq \perp$ holds in all configurations between $\gamma_{R(e,i,2)}$ and $\gamma_{R(e,i,6)}$. Similarly to the previous case, t releases the token by executing $RlT\text{-}action$ within two rounds from $\gamma_{R(e,i,2)}$, a contradiction.

(b) Assume $\gamma_{R(e,i,2)}(t).status = \mathsf{Out}$. If $t.req = \perp$, then $t.req = \perp$ holds in all configurations between γ_i and $\gamma_{R(e,i,6)}$, by hypothesis. Similarly to the previous case, t releases the token by executing $RlT\text{-}action$ within two rounds from $\gamma_{R(e,i,2)}$, a contradiction.

Assume now that $t.req \neq \perp$ in $\gamma_{R(e,i,2)}$. Then, by hypothesis, $t.req \neq \perp$ holds in all configurations between $\gamma_{R(e,i,2)}$ and $\gamma_{R(e,i,6)}$. Moreover, $RsT\text{-}action$ is disabled at t in all configurations between $\gamma_{R(e,i,2)}$ and $\gamma_{R(e,i,6)}$, by Claim 1. Hence, t sets $t.status$ to Wait by $R\text{-}action$ within two rounds from $\gamma_{R(e,i,2)}$ (Remark 1). Then, by Claim 1, $t = Winner(t)$ in all subsequent configurations until $\gamma_{R(e,i,6)}$. After executing $R\text{-}action$, if $IsBlocked(t)$, then by Claim 1, there exists $q \in \mathcal{CN}_p(\gamma)$ such that $q.status = \mathsf{In}$ and $q.req \neq t.req$. By hypothesis, $q.status = \mathsf{In}$ and $q.req \neq t.req$ until at least $\gamma_{M(e,i)}$. So, within at most two more rounds (Remark 1), $t.status$ is set to $\mathsf{Blocked}$ and $t.status$ does not change until at least $\gamma_{M(e,i)}$ (with $R(e,i,6) \leq M(e,i)$) due to q, hence $\gamma_{R(e,i,4)}(t).status = \mathsf{Blocked}$, a contradiction. Assume otherwise that $IsBlocked(t)$ does not hold after t executes $R\text{-}action$. $E\text{-}action$ is continuously enabled at t until t executes it, since by Claim 1 $t = Winner(t)$ in all configurations until $\gamma_{R(e,i,6)}$. So, t executes $E\text{-}action$ within two rounds (Remark 1), and by this latter action, t releases the token by $PassToken(t)$, a contradiction.

(c) Assume $\gamma_{R(e,i,2)}(t).status = \mathsf{Wait}$. We obtain a contradiction similarly to the second part of the previous case.

(d) Assume $\gamma_{R(e,i,2)}(t).status = \mathsf{Blocked}$. We obtain a contradiction similarly to the second part of Case (b).

Claim 3. $IsBlocked(t)$ in all configurations between $\gamma_{R(e,i,4)}$ and $\gamma_{M(e,i)}$.
Proof of the claim: Assume first there a configuration γ_b between $\gamma_{R(e,i,2)}(t)$ and $\gamma_{R(e,i,4)}(t)$ such that $IsBlocked(\gamma_b(t))$. Then, $IsBlocked(\gamma_b(t))$ implies $\neg ResourceFree(\gamma_b(t))$, by Claim 1. Now, by hypothesis, no process ends its critical section until at least $\gamma_{M(e,i)}$. So, $IsBlocked(t)$ holds in all configurations between γ_b and $\gamma_{M(e,i)}$, and we are done.

Assume otherwise that $\neg IsBlocked(t)$ in every configuration between $\gamma_{R(e,i,2)}(t)$ and $\gamma_{R(e,i,4)}(t)$. t cannot executes B-$action$ during $\gamma_{R(e,i,2)}(t) \dots \gamma_{R(e,i,4)}(t)$. So, if $\gamma_{R(e,i,2)}(t).status \neq \mathsf{Blocked}$, then $\gamma_{R(e,i,4)}(t).status \neq \mathsf{Blocked}$, contradicting Claim 2. Otherwise, $Unblock(t)$ holds in every configuration between $\gamma_{R(e,i,2)}(t)$ and $\gamma_{R(e,i,4)}(t)$ until $t.status = \mathsf{Wait}$. Then, as RsT-$action$ is disabled at t in all configurations between $\gamma_{R(e,i,2)}$ and $\gamma_{R(e,i,6)}$ (Claim 1), t switches $t.status$ to Wait by UB-$action$ before $\gamma_{R(e,i,4)}$ (Remark 1) and again $t.status$ remains equal to Wait until at least $\gamma_{R(e,i,4)}$, contradicting Claim 2.

By Claims 2 and 3, $t.status = \mathsf{Blocked}$ in all configurations between $\gamma_{R(e,i,4)}$ and $\gamma_{M(e,i)}$. By Claims 1 and 3, and the hypotheses of the lemma, there is a neighbor q of p such that $q.req \not\simeq t.req$ and $q.status = \mathsf{In}$ in all configurations γ_k between $\gamma_{R(e,i,4)}$ and $\gamma_{M(e,i)}$. By definition, $q \in \mathcal{CN}_t(\gamma_k)$, $\gamma_k(t).req \neq \bot$, and $\gamma_k(q).req \neq \bot$. Finally, by definition of P_{Free}, $\forall p \in P_{Free}(\gamma_k) \cap \mathcal{CN}_t(\gamma_k)$, $p \notin \{q\} \cup \mathcal{CN}_q(\gamma_k)$, indeed no process in P_{Free} can be neighbor of a requesting process with status In (hence in critical section) using a conflicting resource. □

Lemma 14. *Let $e = (\gamma_j)_{j\geq 0}$ be an execution of $\mathcal{LRA} \circ \mathcal{TC}$, $i \geq 0$ such that $\mathcal{TC}$ is stabilized in γ_i (i is defined by Lemma 5), and $t \in V$ the unique tokenholder in γ_i. If $R(e,i,4n+2)$ exists and $R(e,i,4n+2) \leq M(e,i)$ and $\forall j \in \{i+1,\dots,R(e,i,4n+2)\}$, $PassToken(t)$ is not executed in step $\gamma_{j-1} \mapsto \gamma_j$, then for all $k \in \{R(e,i,4n+2),\dots,M(e,i)\}$,*

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

Proof. Let $e = (\gamma_j)_{j\geq 0}$ be an execution of $\mathcal{LRA} \circ \mathcal{TC}$. Let $i \geq 0$ such that $\mathcal{TC}$ is stabilized in γ_i. Let $t \in V$ the unique tokenholder in γ_i. Assume that $R(e,i,4n+2)$ exists, $R(e,i,4n+2) \leq M(e,i)$, and $\forall j \in \{i+1,\dots,R(e,i,4n+2)\}$, $PassToken(t)$ is not executed in step $\gamma_{j-1} \mapsto \gamma_j$.

Claim 1. $\forall j \in \{R(e,i,2),\dots,R(e,i,4n+2)\}$, $\forall p \in V$, $\gamma_j(t).token = \mathtt{true}$ if and only if $p = t$, and RsT-$action$ is disabled at p in γ_j.
Proof of the claim: Identical to the proof of Claim 1 in Lemma 13.

Then, P_{Free} contains only requesting processes p ($p.req \neq \bot$) with no neighbor q using a resource conflicting with the requested one (namely, such that $q.status = \mathsf{In}$ and $q.req \not\simeq p.req$). So, no process can enter P_{Free} during $\gamma_i \dots M(e,i)$ since no new request occurs and no critical section is released.

Let $j \in \{R(e,i,2),\dots,R(e,i,4(n-1)+2)\}$. If $P_{Free}(\gamma_j) \setminus \mathcal{CN}_t(\gamma_j)$ is empty, then it remains so until $\gamma_{R(e,i,4n+2)}$. Otherwise, let $q = \max\{x \in P_{Free}(\gamma_j) \setminus \mathcal{CN}_p(\gamma_j)\}$. In the worst case, q has status Out, it reaches status Wait in at most 2 rounds (by Claim 1 and Remark 1). 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 (using E-$action$) in at most 2 additional rounds (by Claim 1 and Remark 1). Indeed, in the latter case, $IsBlocked(q)$ does not hold since $q \in P_{Free}$ ensures that $ResourceFree(q)$ and since it has no conflicting 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 ensures that in configuration $\gamma_{R(e,i,4n+2)}$ the set $P_{Free}(\gamma_{R(e,i,4n+2)}) \setminus \mathcal{CN}_t(\gamma_{R(e,i,4n+2)})$ is empty. Then, as long as no critical section is released and no new request occurs, P_{Free} remains empty. □

Lemma 15. *Let $e = (\gamma_j)_{j\geq 0}$ be an execution of $\mathcal{LRA} \circ \mathcal{TC}$ and $i \geq 0$ such that $\mathcal{TC}$ is stabilized in γ_i (i is defined by Lemma 5). 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 $k \in \{R(e,i,6n(N_{tok}+1)),\dots,M(e,i)\}$, $P_{Free}(\gamma_k) = \emptyset$, or*
- *for every $k \in \{R(e,i,6n(N_{tok}+1)-6),\dots,M(e,i)-1\}$, $PassToken$ is not executed in step $\gamma_k \mapsto \gamma_{k+1}$.*

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

Similarly to the proof of Lemma 14, P_{Free} cannot increase, hence if it is empty at some configuration γ_k with $k \in \{R(e, i, 6n(N_{tok} + 1)), \ldots, M(e, i)\}$, we are done.

Let $k \in \{R(e, i, 6n(N_{tok} + 1)), \ldots, M(e, i)\}$. Assume $P_{Free}(\gamma_k) \neq \emptyset$ and let $p \in P_{Free}(\gamma_k)$. We deduce from Lemma 13, that if $PassToken$ has not been executed by the tokenholder, 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 additional rounds (still by Lemma 13), 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 16. *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 $e = (\gamma_i)_{i\geq 0}$ 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 $T_{PC} = T_{tok} + 6n(N_{tok} + 1) + 4n - 4$. Let $e = (\gamma_i)_{i\geq 0}$ 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 15, we have two cases:

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

2. For every $k \in \{R(e, i, 6n(N_{tok} + 1) - 6), \ldots, M(e, i) - 1\}$, $PassToken$ is not executed in step $\gamma_k \mapsto \gamma_{k+1}$. Note that this implies that $PassToken$ is not executed during the last 6 rounds by the tokenholder t: this allows to apply Lemma 13: there exists a conflicting neighbor of t, q, such that $\forall p \in P_{Free} \cap \mathcal{CN}_t(\gamma_k)$, $p \notin \{q\} \cup \mathcal{CN}_q(\gamma_k)$.

   As t holds the token from configuration $R(e, i, 6n(N_{tok}+1)-6)$ to configuration $M(e, i)$, and as $R(e, i, T_{tok}+6n(N_{tok}+1)+4n-4) \leq M(e, i)$, we can apply Lemma 14 between configuration $R(e, i, T_{tok} + 6n(N_{tok} + 1) - 6)$ and $R(e, i, T_{tok} + 6n(N_{tok} + 1) + 4n - 4)$: this proves that $P_{Free}(\gamma_{R(e,i,T_{tok}+6n(N_{tok}+1)+4n-4)}) \setminus \mathcal{CN}_t(\gamma_{R(e,i,T_{tok}+6n(N_{tok}+1)+4n-4)})$ is empty.

□

By Lemmas 12 and 16, follows.

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 [9] 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.[5] Now, perhaps surprisingly, our results show that *maximal concurrency* cannot be achieved in problems that can be expressed with the LRA paradigm. However, we

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

showed that *strong 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 [4]. Defining the exact class of resource allocation problems where *maximal concurrency* (resp. *strong concurrency*) can be achieved is a challenging perspective.

References

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] 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.

[5] 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.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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

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

[16] Shlomi Dolev. *Self-stabilization*. MIT Press, March 2000.

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

[18] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Uniform Dynamic Self-Stabilizing Leader Election. *IEEE Trans. Parallel Distrib. Syst.*, 8(4):424–440, 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] Leslie Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Commun. ACM*, 17(8):453–455, 1974.

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

[26] 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.