Concurrency in Snap-Stabilizing Local Resource Allocation

Karine Altisen, Stéphane Devismes, Anaïs Durand

To cite this version:

Karine Altisen, Stéphane Devismes, Anaïs Durand. Concurrency in Snap-Stabilizing Local Resource Allocation. [Research Report] VERIMAG. 2014. hal-01099186v2

HAL Id: hal-01099186
https://hal.science/hal-01099186v2

Submitted on 9 Jan 2015 (v2), last revised 17 Nov 2016 (v8)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Concurrency in Snap-Stabilizing Local Resource Allocation

Karine Altisen, Stéphane Devismes, and Anaïs Durand

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

Abstract

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

1 Introduction

Mutual exclusion [13, 23] 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 [18] 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 [25] 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.

Thus, for efficiency purposes, algorithms solving such problems must be as parallel as possible. As a consequence, these algorithms should be, in particular, evaluated at the light of the level of concurrency they permit, and this level of concurrency should be captured by a dedicated property. However, most of the solutions to resource allocation problems simply do not consider the concurrency issue, *e.g.*, [4, 6, 8, 19, 20, 22, 24]

Now, as quoted by Fischer *et al.* [18], specifying resource allocation problems without including a property of concurrency may lead to degenerated solutions, *e.g.*, any mutual exclusion algorithm realizes the safety and the fairness of ℓ-exclusion. To address this issue, Fischer *et al.* [18] 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 meanwhile". Some other properties, inspired from the avoiding ℓ-deadlock property, have been proposed to capture the level of concurrency in other resource allocation problems, *e.g.*, k-out-of-ℓ-exclusion [10] and committee coordination [5]. However, until now, all existing properties of concurrency are specific to a particular problem.

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

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

We propose a LRA algorithm achieving (strong) partial maximal-concurrency in bidirectional identified networks of arbitrary topology. As additional feature, this algorithm is *snap-stabilizing* [7]. *Snap-stabilization* is a versatile property which enables a distributed system to efficiently withstand transient faults. Informally, after transient faults cease, a snap-stabilizing algorithm *immediately* resumes correct behavior, without external intervention. More precisely, a snap-stabilizing algorithm guarantees that any computation started after the faults cease will operate correctly. However, we have no guarantees for those executed all or a part during faults. By definition, snap-stabilization is a strengthened form of *self-stabilization* [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.*, [4, 6, 8, 19, 20, 22, 24]. In [6], Boulinier *et al.* propose a self-stabilizing unison algorithm which allows to solve local mutual exclusion, local group mutual exclusion, and the local reader-writers problem. There are also many self-stabilizing algorithms for the local mutual exclusion [4, 19, 22, 24]. In [20], Huang proposes a self-stabilizing algorithm solving the dining philosophers problem. A self-stabilizing drinking philosophers algorithm is given in [24]. In [8], Cantarell *et al.* generalizes the above problems by introducing LRA and illustrates it with a self-stabilizing algorithm. To the best of our knowledge, no other paper deals with the general instance of LRA and no paper proposes snap-stabilizing solution for any particular instance of LRA. Finally, none of the aforementioned papers (especially [8]) consider the concurrency issue.

Roadmap. The next section introduce the computation model and the specification of the LRA problem. In Section 3, we introduce the property of maximal-concurrency, show the impossibility result, and introduce the property of partial maximal-concurrency. Our algorithm is presented in Section 4.

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

We prove its correctness in Section 5 and its partial maximal-concurrency in Section 6. We conclude in Section 7.

2 Computational Model and Specifications

2.1 Distributed Systems

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

2.2 Locally Shared Memory Model

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

2.3 Snap-Stabilizing Local Resource Allocation

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

2.3.1 Local Resource Allocation

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

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

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

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

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. Let $\mathcal{E}$ be the set of all possible executions of $\mathcal{A}$. A *specification* SP is a predicate over $\mathcal{E}$. In [7], snap-stabilization has been defined as follows: $\mathcal{A}$ is *snap-stabilizing w.r.t.* SP if $\forall e \in \mathcal{E}$, e satisfies SP.

Of course, not all specifications — in particular their safety part — can be satisfied when considering a system which can start from an arbitrary configuration. Actually, snap-stabilization's notion of safety

is *user-centric*: when the user initiates a computation, then the computed result should be correct. So, we express a problem using a *guaranteed service specification* [2]. Such a specification consists in specifying three properties related to the computation start, computation end, and correctness of the delivered result. (In the context of LRA, this latter property will be referred to as "resource conflict freedom".)

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

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

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

Definition 1 (Snap-stabilizing Local Resource Allocation)**.** $\mathcal{A}$ is a *snap-stabilizing local resource allocation* algorithm if $\forall e = (\gamma_i)_{i \geq 0} \in \mathcal{E}$, e satisfies the guaranteed service specification SP defined by the three following properties:

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):

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

3 Concurrency

Most of existing resource allocation algorithms, especially self-stabilizing ones [4, 6, 8, 19, 20, 22, 24], do not consider the concurrency issue. In [18], authors propose a concurrency property *ad hoc* to ℓ-exclusion. We now define the *maximal-concurrency*, which generalizes the definition of [18] to any resource allocation problem.

3.1 Maximal-Concurrency

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

Let $P_{CS}(\gamma)$ be the set of processes that are executing their critical section in γ, *i.e.*, the set of processes holding resources in γ. Let $P_{Req}(\gamma)$ be the set of processes that are requesting 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. We denote by $\gamma(p).req$ the resource(s) requested by process p in γ. Let

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

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

$$\big(continuousCS(\gamma_i \ldots \gamma_{i+j}) \wedge P_{Free}(\gamma_i) \neq \emptyset\big) \Rightarrow \quad \big(\exists k \in \{i, \ldots, i+j-1\}, \exists p \in V, \\ p \in P_{Free}(\gamma_k) \cap P_{CS}(\gamma_{k+1})\big)$$

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

Example 1: ℓ-Exclusion Maximal-Concurrency. In the ℓ-exclusion problem, up to ℓ processes can execute their critical section concurrently . 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.* [18].

Example 2: Local Resource Allocation Maximal-Concurrency. In the local resource allocation problem, a requesting process is allowed to enter its critical section if all its neighbors in critical section are using resources which are compatible with its request:

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

The maximal-concurrency property can also be defined as follows:

Definition 3 (Maximal Concurrency)**.** An algorithm is *maximal concurrent* if and only if $\forall e = (\gamma_i)_{i \geq 0} \in \mathcal{E}, \forall i \geq 0, \exists T \in \mathbb{N}, \forall t \geq T$,

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

Definition 2 and Definition 3 are equivalent using induction arguments.

Lemma 1. *Definition 2 and Definition 3 are equivalent.*

Proof. Let $e = (\gamma_i)_{i \geq 0} \in \mathcal{E}$, $i \geq 0$.

Assume Definition 2 holds. If $P_{Free}(\gamma_i)$ is not empty, there exists some $N \in \mathbb{N}$ such that if $continuousCS$ continuously holds between γ_i and γ_{i+N+1} the set P_{Free} strictly decreases meanwhile (as it cannot increase). Applying inductively this step of reasoning and as $P_{Free}(\gamma_i)$ is finite proves that within a finite number of steps P_{Free} becomes empty provided that $continuousCS$ continuously holds.

Conversely, assume that for some $T \in \mathbb{N}$, for all $t \geq T$, $continuousCS(\gamma_i \dots \gamma_{i+t}) \implies P_{Free}(\gamma_{i+t}) = \emptyset$. If $P_{Free}(\gamma_i) = \emptyset$, we are done. Otherwise, again the set P_{Free} can never increase from γ_i to γ_{i+T}. But, from γ_i to γ_{i+t}, it passes from non-empty to empty: this means it strictly decreases meanwhile. We pose $N \in \mathbb{N}$ the maximum number of steps between two processes going out P_{Free}: with such N, the proposition of Definition 2 becomes true. □

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

3.2 Maximal Concurrency vs. Fairness

Definition 4 below gives a definition of fairness classically used in resource allocation problems. Notably, Computation Start and End properties of Definition 1 trivially implies this fairness property. Next, Theorem 1 states that no LRA algorithm can achieve maximal-concurrency. Actually, its proof is based on the incompatibilty between fairness and maximal-concurrency.

Definition 4 (Fairness)**.** Each time a process is (continuously) requesting a resource r, it eventually accesses r.

Theorem 1. *It is impossible to design a local resource allocation algorithm (stabilizing or not) for arbitrary networks that satisfies maximal-concurrency.*

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

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

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

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

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

3.3 Partial Maximal-Concurrency

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

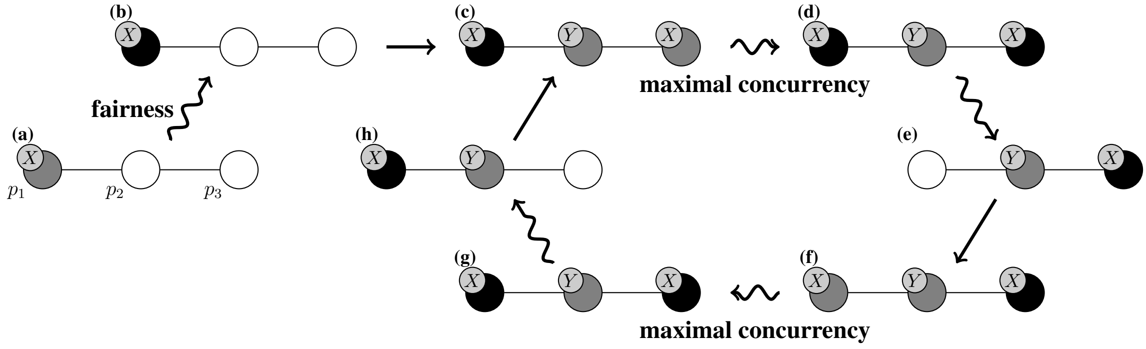


Figure 1: Maximal concurrency vs. fairness. The processes in black are executing their critical section. The processes in gray are requesting resources. The processes in white are idle. Requested resources are given in the bubbles next to the nodes.

Definition 5 (Partial Maximal-Concurrency)**.** An algorithm $\mathcal{A}$ is *partially maximal-concurrent* if and only if $\forall e = (\gamma_i)_{i \geq 0} \in \mathcal{E}$, $\forall i \geq 0$, $\exists T \in \mathbb{N}$ such that $\forall t \geq T$, $\exists X \subseteq V$ such that

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

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

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

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

Definition 6 (Strong Partial Maximal-Concurrency)**.** An algorithm $\mathcal{A}$ is *strongly partially maximal-concurrent* if and only if $\forall e = (\gamma_i)_{i \geq 0} \in \mathcal{E}$, $\forall i \geq 0$, $\exists T \in \mathbb{N}$ such that $\forall t \geq T$, $\exists p, q \in V$, $q \in \mathcal{N}_p$ such that

$$continuousCS(\gamma_i \dots \gamma_{i+t}) \Rightarrow P_{Free}(\gamma_{i+t}) \subseteq \mathcal{N}_p \backslash \{q\}$$

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

4 Local Resource Allocation Algorithm

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

4.1 Composition

These two modules are composed using a *fair composition* [16], denoted $\mathcal{LRA} \circ \mathcal{TC}$. In such a composition, each process executes a step of each algorithm alternately.

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

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

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

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

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

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

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

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

$$lbl_1 : (b = 1) \wedge \neg grd_1^A \wedge \neg grd_2^A \wedge \cdots \wedge \neg grd_x^A \to b \leftarrow 2$$

$$lbl_2 : (b = 2) \wedge \neg grd_1^B \wedge \neg grd_2^B \wedge \cdots \wedge \neg grd_y^B \to b \leftarrow 1$$

4.2 Token Circulation Module

We assume that $\mathcal{TC}$ is a self-stabilizing black box which allows $\mathcal{LRA}$ to emulate a self-stabilizing token circulation. $\mathcal{TC}$ provides two outputs to each process p in $\mathcal{LRA}$: the predicate $Token(p)$ and the statement $PassToken(p)$. The predicate $Token(p)$ expresses whether the process p holds a token or not. 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 $Token(p)$ holds. Precisely, we assume that $\mathcal{TC}$ satisfies the following properties.

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

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

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

To design $\mathcal{TC}$ we proceed as follows. There exist several self-stabilizing token circulations for arbitrary rooted networks [9, 11, 21] that contain a particular action, $T : Token(p) \to PassToken(p)$, 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 $Token(p)$ and $PassToken(p)$ as outputs, for every process p.

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
$Token(p)$: Predicate from $\mathcal{TC}$, indicate that p holds the token
$PassToken(p)$: Statement from $\mathcal{TC}$, pass the token to a neighbor

Macros

$WaitingNeigh(p)$	$\equiv$	$\{q \in \mathcal{N}_p \mid q.status = \mathsf{Wait}\}$
$LocalMax(p)$	$\equiv$	$\max\{q \in WaitingNeigh(p) \cup \{p\}\}$
$LocalTokens(p)$	$\equiv$	$\{q \in \mathcal{N}_p \cup \{p\} \mid q.token\}$
$TokenMax(p)$	$\equiv$	$\max\{q \in LocalTokens(p)\}$

Predicates

$ResourceFree(p)$	$\equiv$	$\forall q \in \mathcal{N}_p, \big(q.status = \mathsf{In} \Rightarrow p.req \rightleftharpoons q.req\big)$
$IsBlocked(p)$	$\equiv$	$\neg ResourceFree(p) \vee \big(\exists q \in \mathcal{N}_p, q.status = \mathsf{Blocked} \wedge q.token\big)$
$TokenAccess(p)$	$\equiv$	$LocalTokens(p) \neq \emptyset \wedge p = TokenMax(p)$
$MaxAccess(p)$	$\equiv$	$LocalTokens(p) = \emptyset \wedge p = LocalMax(p)$

Guards

$Requested(p)$	$\equiv$	$p.status = \mathsf{Out} \wedge p.req \neq \bot$
$Block(p)$	$\equiv$	$p.status = \mathsf{Wait} \wedge IsBlocked(p)$
$Unblock(p)$	$\equiv$	$p.status = \mathsf{Blocked} \wedge \neg IsBlocked(p)$
$Enter(p)$	$\equiv$	$p.status = \mathsf{Wait} \wedge \neg IsBlocked(p) \wedge (TokenAccess(p) \vee MaxAccess(p))$
$Exit(p)$	$\equiv$	$p.status = \mathsf{In} \wedge p.req = \bot$
$ResetToken(p)$	$\equiv$	$Token(p) \neq p.token$
$ReleaseToken(p)$	$\equiv$	$p.token \wedge p.status \in \{\mathsf{Out}, \mathsf{In}\}$

Actions

(1)	*RsT-action*	::	$ResetToken(p)$	$\rightarrow$	$p.token \leftarrow Token(p)$;
(2)	*RlT-action*	::	$ReleaseToken(p)$	$\rightarrow$	$PassToken(p)$;
(3)	*R-action*	::	$Requested(p)$	$\rightarrow$	$p.status \leftarrow \mathsf{Wait}$;
(3)	*B-action*	::	$Block(p)$	$\rightarrow$	$p.status \leftarrow \mathsf{Blocked}$;
(3)	*UB-action*	::	$Unblock(p)$	$\rightarrow$	$p.status \leftarrow \mathsf{Wait}$;
(3)	*E-action*	::	$Enter(p)$	$\rightarrow$	$p.status \leftarrow \mathsf{In}$; **if** $p.token$ **then** $PassToken(p)$ **fi**;
(3)	*Ex-action*	::	$Exit(p)$	$\rightarrow$	$p.status \leftarrow \mathsf{Out}$;

4.3 Resource Allocation Module

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

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

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

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

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

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

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

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

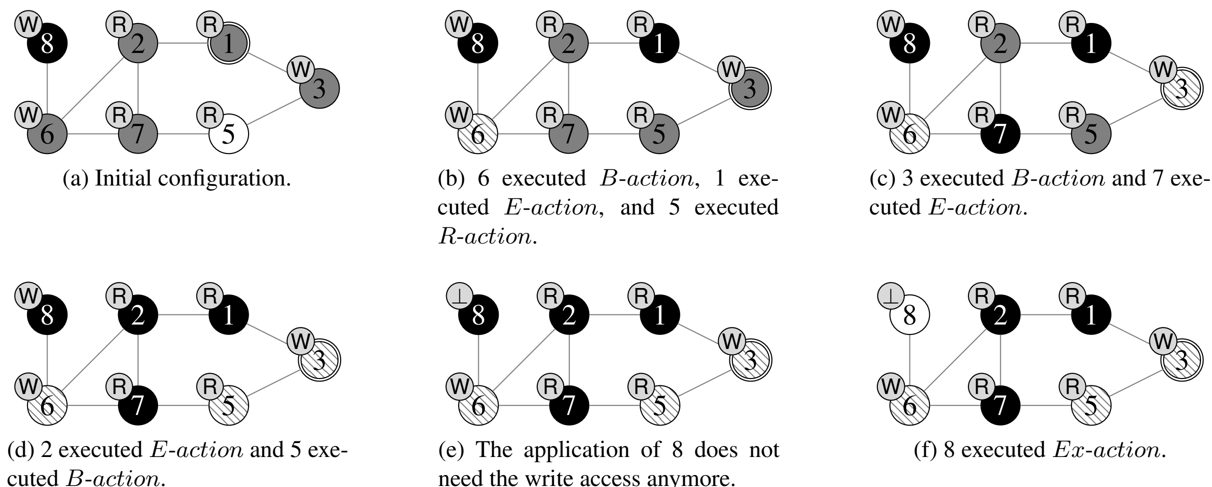
The predicate $Start(\gamma_i, \gamma_{i+1}, p, r)$ becomes true when process p takes the request for resource r into account in $\gamma_i \mapsto \gamma_{i+1}$, *i.e.*, when the status of p switches from Out to Wait in $\gamma_i \mapsto \gamma_{i+1}$ because $p.req = r \neq \bot$ in γ_i.

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

We now illustrate the principles of $\mathcal{LRA}$ with the example given in Figure 2. In this example, we consider the local reader-writer problem. In the figure, the numbers inside the nodes represent their IDs. The color of a node represents its status: white for Out, gray for Wait, black for In, and crossed out for Blocked. A double circled node holds a token. The bubble next to a node represents its request. Recall that we have two resources: R for a reading access and W for a writing access.

When the process is idle ($p.status = \mathsf{Out}$), its application can request a resource. In this case, $p.req = r \neq \bot$ and p sets $p.status$ to Wait by $R\text{-}action$: p starts the computation to obtain r. For example, 5 starts a computation to obtain R in (a)$\mapsto$(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)$\mapsto$(b)). If there is no more neighbors 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 $\big(p.token \wedge \neg q.token\big) \vee \big(p.token = q.token \wedge p > q\big)$. More precisely, if there is no token in the neighborhood of



(a) Initial configuration.

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

(c) 3 executed B-$action$ and 7 executed E-$action$.

(d) 2 executed E-$action$ and 5 executed B-$action$.

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

(f) 8 executed Ex-$action$.

Figure 2: Example of execution of $\mathcal{LRA} \circ \mathcal{TC}$.

p, the highest priority process is the waiting process with highest ID. Otherwise, the token holders (there may be several tokens during the stabilization phase of $\mathcal{TC}$) blocked all their requesting neighbors, even if they request for non-conflicting resources, and until the token holders obtain their requested resources. This mechanism allows to ensure fairness by slightly decreasing the level of concurrency. (The token circulates to eventually give priority to blocked processes, *e.g.*, processes with small IDs.)

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

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

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

Hence, a requesting process is served in a finite time. This is illustrated by an example of execution on Figure 3. We consider here the local mutual exclusion problem in which two neighbors cannot concurrently execute their critical section. We try to delay as long as possible the service of process 2. As its neighbors 7 and 8 also request the resource but have greater IDs, they can access their critical section before 2 (see Steps (a)$\mapsto$(b) and (e)$\mapsto$(f)). But a token circulates in the network and eventually reaches 2 (see Configuration (g)). Then, 2 has priority over its neighbors (even if it has a lower ID) and eventually starts executing its critical section in (j)$\mapsto$(k).

5 Correctness of $\mathcal{LRA} \circ \mathcal{TC}$

In this section, we show that $\mathcal{LRA} \circ \mathcal{TC}$ satisfies the snap-stabilizing specification of the local resource allocation problem (Definition 1). The predicates of this specification are instantiated as follow:

- $Request(\gamma, p, r) \equiv \gamma(p).req = r$

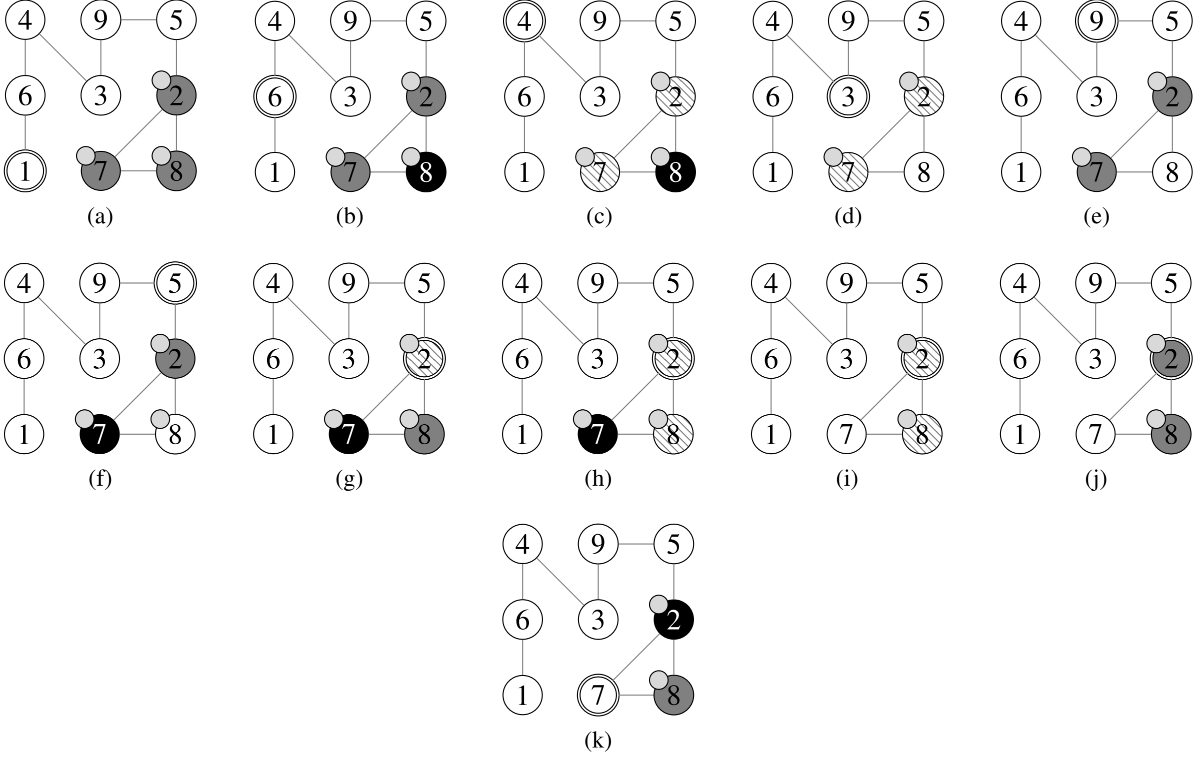


Figure 3: Example of execution of $\mathcal{LRA} \circ \mathcal{TC}$ on the local mutual exclusion problem. The bubbles mark the requesting processes.

- $Start(\gamma_i, \gamma_{i+1}, p, r) \equiv \gamma_i(p).status = \mathsf{Out} \wedge \gamma_{i+1}(p).status = \mathsf{Wait} \wedge \gamma_i(p).req = \gamma_{i+1}(p).req = r$
- $Result(\gamma_i \ldots \gamma_j, p, r) \equiv \gamma_i(p).status = \mathsf{Wait} \wedge \gamma_{i+1}(p).status = \mathsf{In} \wedge \gamma_i(p).req = \gamma_{i+1}(p).req = r \wedge \gamma_{j-1}(p).status = \mathsf{In} \wedge \gamma_j(p).status = \mathsf{Out}$
- $NoConflict(\gamma, p) \equiv \gamma(p).status = \mathsf{In} \Rightarrow \big(\forall q \in \mathcal{N}_p, \gamma(q).status = \mathsf{In} \Rightarrow (\gamma(q).req \rightleftharpoons \gamma(p).req)\big)$

First, we show that the resource conflict freedom property is always satisfied.

Remark 1. If *E-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 2. *E-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-action* is enabled at p and q in γ. Then, $\big(TokenAccess(p) \vee MaxAccess(p)\big)$ holds in γ. Let examine these two cases:

1. If $TokenAccess(p)$ holds in γ, $p = TokenMax(p)$ in γ. So, $\gamma(p).token = \texttt{true}$, and, consequently $p \in LocalTokens(q) \neq \emptyset$ in γ. Hence, $\neg MaxAccess(q)$ in γ and then $TokenAccess(q)$ holds in γ. (Otherwise, *E-action* is not enabled at q in γ.)

   Now, $p = TokenMax(p) = \max\{x \in LocalTokens(p)\} > q$ since $q \in LocalTokens(p)$ and $q = TokenMax(q) = \max\{x \in LocalTokens(q)\} > p$ since $p \in LocalTokens(q)$, a contradiction.

2. If $MaxAccess(p)$ holds in γ, $LocalTokens(p) = \emptyset$ in γ. As $q \in \mathcal{N}_p$, $\gamma(q).token = \texttt{false}$ so $q \neq TokenMax(q)$ in γ, and, consequently, $\neg TokenAccess(q)$ in γ. Hence, $MaxAccess(q)$ holds in γ. (Otherwise, $E\text{-}action$ is not enabled at q in γ.)

   Now, $p = LocalMax(p) = \max\{x \in WaitingNeigh(p) \cup \{p\}\} > q$ since $q \in WaitingNeigh(p)$ and $q = LocalMax(q) = \max\{x \in WaitingNeigh(q) \cup \{q\}\} > p$ since $p \in WaitingNeigh(q)$, a contradiction.

□

Lemma 3. *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 $\gamma'(q).req \not\rightleftharpoons \gamma'(p).req$.

Using Property 4, there is only two way to change the value of $p.req$ between γ and γ':

- From some $r \in \mathcal{R}_p$, $r \neq \bot$, to $\bot$: As $\bot$ is compatible with every resource, if $\gamma'(p).req = \bot$, $\gamma'(p).req \rightleftharpoons \gamma'(q).req$, a contradiction.
- From $\bot$ to some $r \in \mathcal{R}_p$: The application of p can only change its request this way if $\gamma(p).status = \mathsf{Out}$. But $\gamma'(p).status = \mathsf{In}$ and there is no way to go from Out to In in one step.

Hence, $\gamma(p).req = \gamma'(p).req$. We can make the same reasoning on q so $\gamma(q).req = \gamma'(q).req$.

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, $\gamma(q).status \neq \mathsf{In}$. (Otherwise, as $\gamma(q).req \not\rightleftharpoons \gamma(p).req$, we have $\neg NoConflict(\gamma, p)$.) So q executes an action during $\gamma \mapsto \gamma'$ to obtain $status$ In. The only possible action is $E\text{-}action$. So q executes $E\text{-}action$ even if it has a neighbor using a conflicting resource, a contradiction to Remark 1.
2. If $\gamma(p).status \neq \mathsf{In}$, then p execute an action in $\gamma \mapsto \gamma'$. p can only execute $E\text{-}action$ in $\gamma \mapsto \gamma'$ (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 2.
   (b) If $\gamma(q).status = \mathsf{In}$, then $E\text{-}action$ is enabled at p in γ even though a neighbor of p has $status$ In and a conflicting request (p is in a similar situation to the one of q in case 1), a contradiction to Remark 1.

□

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. Let $k \geq 0$ and $k' > k$. Let $p \in V$. Let $r \in \mathcal{R}_p$. Assume $Result(\gamma_k \ldots \gamma_{k'}, p, r)$. 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 3, $\forall i \geq l$, $NoConflict(\gamma_i, p)$ holds. In particular, $\forall i \in \{k, \ldots, k'\}$, $NoConflict(\gamma_i, p)$. □

In the following, we assume a weakly fair daemon.

Lemma 4. *The stabilization of $\mathcal{TC}$ is preserved by fair composition.*

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

In the following, we will consider that $\mathcal{TC}$ has stabilized.

Lemma 5. *A process cannot keep a token forever in $\mathcal{LRA} \circ \mathcal{TC}$.*

Proof. Let e be an infinite execution. By Lemma 4, the token circulation eventually stabilizes, *i.e.*, there is a unique token in every configuration after the stabilization of $\mathcal{TC}$. Assume by contradiction that, after such a configuration γ, a process p keeps the token forever: $Token(p)$ holds forever and $\forall q \in V$, $q \neq p$, $\neg Token(q)$ holds forever.

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

Then, there are three cases:

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

2. If $\gamma'(p).status = \mathsf{Wait}$, $TokenAccess(p)$ holds forever and $\forall q \in \mathcal{N}_p$, $\neg TokenAccess(q)$ holds forever. So, $\forall q \in \mathcal{N}_p$, $E\text{-}action$ is disabled forever. Now, if $\exists q \in \mathcal{N}_p$ such that $\gamma'(q).status = \mathsf{In} \wedge \gamma'(q).req \not\asymp \gamma'(p).req$, then, as $\perp$ is compatible with any resource, $\gamma'(q).req \neq \perp$. Using Property 5, in finite time the request of q becomes $\perp$ and remains $\perp$ until q obtains $status$ Out (Property 4). So $RequestOut(p)$ continuously holds, and $Ex\text{-}action$ is continuously enabled at q. Hence, in finite time, $\forall q \in \mathcal{N}_p$, $q.status \neq \mathsf{In}$ forever. So $E\text{-}action$ is continuously enabled at p and, in finite time, p executes $E\text{-}action$ and releases its token, a contradiction.

3. If $\gamma'(p).status = \mathsf{Blocked}$, $\forall q \in \mathcal{N}_p$, $IsBlocked(q)$ holds forever so $E\text{-}action$ is disabled at q forever. Now, as in case 2, $\forall q \in \mathcal{N}_p$, $q.status \neq \mathsf{In}$ in finite time, and then, $UB\text{-}action$ is continuously enabled at p. in finite time, p gets $status$ Wait and, as in case 2, eventually releases its token, a contradiction.

□

Lemma 5 implies that the hypothesis of Property 3 is satisfied. Hence, we can deduce Corollary 1.

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

Lemma 6. *A process of status* Wait *executes $E\text{-}action$ in finite time.*

Proof. Let e be an infinite execution. By Lemma 4, the token circulation eventually stabilizes. Let $\gamma \in e$ be a configuration of the system after stabilization of $\mathcal{TC}$. Let $p \in V$. Assume that $\gamma(p).status = \mathsf{Wait}$. By Corollary 1, in finite time p holds the unique token. Now, a process cannot keep forever the token (Lemma 5) and p can only release its token executing $E\text{-}action$ (by Property 2). □

Lemma 7. *A process of status* Wait *gets status* Out *in finite time.*

Proof. Let $p \in V$. Let γ be a configuration. If $\gamma(p).status = \mathsf{Wait}$, then p executes $E\text{-}action$ in a finite time (Lemma 6) and gets $status$ In in configuration γ'. Now, if $\gamma'(p).req \neq \bot$, in finite time it changes to $\bot$ (Property 5) and cannot change anymore until p gets $status$ Out (Property 4). So, $RequestOut(p)$ continuously holds and $Ex\text{-}action$ is continuously enabled at p. So, p executes $Ex\text{-}action$ and eventually gets $status$ Out. □

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

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

Proof. Let $e = (\gamma_i)_{i \geq 0}$ be an execution. Let $k \geq 0$. Let $p \in V$. Let $r \in \mathcal{R}_p$. First, p eventually has $status$ Out. Indeed, if $\gamma_k(p).status \neq \mathsf{Out}$, p is computing a service and cannot answer a new request (the application cannot change its request before being served by Property 4). But, in finite time, p ends its current computation and obtains $status$ Out (Lemma 7), let say in $\gamma_{j-1} \mapsto \gamma_j$ ($j \geq k$).

Now, if $\gamma_j(p).req \neq \bot$ holds, it holds continuously (Property 4). Hence, $R\text{-}action$ is continuously enabled at p, and, p eventually executes $R\text{-}action$ (let say in $\gamma_l \mapsto \gamma_{l+1}$, $l > j \geq k$). So $\gamma_{l+1}(p).status = \mathsf{Wait}$. Notice that the application of p cannot change its mind (Property 4), so $\gamma_l(p).req = \gamma_{l+1}.req = r$. Hence, $Request(\gamma_l, p, r)$ and $Start(\gamma_l, \gamma_{l+1}, p, r)$ hold. □

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

Proof. Let $e = (\gamma_i)_{i \geq 0}$ be an execution. Let $k \geq 0$. Let $p \in V$. Let $r \in \mathcal{R}_p$. If $Start(\gamma_k, \gamma_{k+1}, p, r)$ holds, then $\gamma_{k+1}(p).status = \mathsf{Wait}$ and $\gamma_{k+1}(p).req = r$. Using Lemma 6, in finite time, p executes $E\text{-}action$ and gets $status$ In (let say in $\gamma_{l-1} \mapsto \gamma_l$, $l > k$). Notice that the application cannot change the value of req until p obtains $status$ In (Property 4) so $\gamma_{l-1}(p).req = \gamma_l(p).req = \gamma_{k+1}(p).req = r$.

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

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

Theorem 5 (Correctness)**.** *Algorithm $\mathcal{LME} \circ \mathcal{TC}$ is a snap-stabilizing local mutual exclusion algorithm working under a weakly fair daemon.*

6 Strong Partial Maximal Concurrency of $\mathcal{LRA} \circ \mathcal{TC}$

We ow show that $\mathcal{LRA} \circ \mathcal{TC}$ is strongly partially maximal-concurrent.

Theorem 6. *Algorithm $\mathcal{LRA} \circ \mathcal{TC}$ is strongly partially maximal-concurrent.*

Proof. Let $e = (\gamma_i)_{i \geq 0}$ be an execution. Let $i \geq 0$. As $\mathcal{TC}$ is self-stabilizing, there exists in this execution e a date $s \geq 0$ after which $\mathcal{TC}$ has stabilized. Let $j \geq s$ and assume that $continuousCS(\gamma_i, ..., \gamma_j)$.

1. Then, for some process $p \in V$, there may exist a date $k \in \{i, ..., j\}$ such that $p \in P_{CS}(\gamma_k)$: then for all $k' \in \{k, ..., j\}$, $p \in P_{CS}(\gamma'_k)$ (by definition of $continuousCS$), $\gamma_{k'}(p).req$ stays unchanged and $\gamma_{k'}(p).status$ is In (see Property 4).

2. For other process $p \in V$, we have that $\forall k \in \{i, ..., j\}$, $p \notin P_{CS}(\gamma_k)$ and

   (a) either $\forall k \in \{i, ..., j\}$, $p \notin P_{Req}(\gamma_k)$ also – in this case, $\gamma_k(p).req = \bot$ and $p.status$ can be In and become Out or can be directly Out;

(b) or $\exists k' \in \{i, ..., j\}$ such that $p \in P_{Req}(\gamma_{k'})$ – in this case, since k', $p.req \neq \bot$ remains unchanged, $p.status$ can be Out, Wait or Blocked.

Now, we can chose $j \geq s$ such that for processes $p \in V$ in case 2,

(a') for sub-case 2a, there exists $k_p \in \{i, ..., j\}$ such that $p.status$ is Out since k_p;

(b') for sub-case 2b, there exists $k_p \in \{i, ..., j\}$ such that since k_p, $p.req$ stays unchanged ($\neq \bot$), $p.status$ is Blocked if $p \notin P_{Free}(\gamma_{k_p})$ and $p.status$ is either Wait or Blocked if $p \in P_{Free}(\gamma_{k_p})$.

This can be easily proved by induction, using the fact that for the corresponding actions, if one becomes true for some process, it remains continuously enabled until, in finite time, the weakly fair daemon selects the process that performs this action.

We pose $T \geq s$ the maximum of the dates k_p as defined above. Therefore for all $j' \geq j$, if $continuousCS(\gamma_i, ..., \gamma_{j'})$ holds, then for all $k \in \{T, ..., j'\}$ the above property (cases 1, 2a' and 2b') holds: this implies that T does not depend on the choice of j. Note also that after T and provided that $continuousCS$ still continuously holds, R-$action$, B-$action$, UB-$action$, E-$action$, and Ex-$action$ are disabled (and P_{Free} remains unchanged).

Let $t \geq T$. We now show that for every process $p \in P_{Free}(\gamma_t)$ (case 2b'), $\gamma_t(p).status = \text{Blocked}$, (*i.e.*, $\exists q \in \mathcal{N}_p$ such that $\gamma_t(q).status = \text{Blocked} \wedge \gamma_t(q).token = \texttt{true}$).

Assume by contradiction that $\gamma_t(p).status = \text{Wait}$. There are two cases:

1. if $p = LocalMax(p)$, E-$action$ is enabled at p, a contradiction to the definition of T;

2. if $q = LocalMax(p)$, $q \neq p$, we can build a sequence of processes $p_0, p_1, \ldots, p_k$ where $p_0 = p$, $p_1 = q$, and such that $\forall i \in \{1, \ldots, k\}$, $p_i = LocalMax(p_{i-1})$. Notice that $\forall i \in \{0, \ldots, k\}$, $p_i \in P_{Free}(\gamma_t)$. This sequence is finite because $p_0 < p_1 < \cdots < p_k$ (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, $p_k = LocalMax(p_k)$ and p_k is in the same case as p in 1, a contradiction.

Let $p \in P_{Free}(\gamma_t)$: p is blocked because of the unique token holder, let say x. Then, $p \in \mathcal{N}_x$ and $P_{Free}(\gamma_t)$ contains all the requesting neighbors of x. In the worst case, it contains all the neighborhood of x except one process $y \in \mathcal{N}_x$ such that $\gamma_t(y).status = \text{In}$ and $\gamma_t(y).req \neq \gamma_t(x).req$, namely, the one that blocks x. Hence, $P_{Free}(\gamma_t) \subseteq \mathcal{N}_x \backslash \{y\}$, and $\mathcal{LRA} \circ \mathcal{TC}$ is strongly partially maximal-concurrent.

□

7 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 [18] and (k,ℓ)-liveness [10] defined for the ℓ-exclusion and k-out-of-ℓ-exclusion, respectively. From [18, 10], we already know that *maximal-concurrency* can be achieved in some important global resource allocation problems.[2] Now, perhaps surprisingly, our results show that *maximal-concurrency* cannot be achieved in problems that can be expressed with the LRA paradigm. However, we showed that *strong partial maximal-concurrency* (an high, but not maximal, level of concurrency) can be achieved by a snap-stabilizing LRA algorithm. We have to underline that the level of concurrency we achieve here is similar to the one obtained in the committee coordination problem [5]. Defining the exact class of resource allocation problems where *maximal-concurrency* (resp. *strong partial maximal-concurrency*) can be achieved is a challenging perspective.

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

References

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

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

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

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

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

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

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

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

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

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

[11] Ajoy Kumar Datta, 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, 2000.

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

[18] 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.

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

[20] 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.

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

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

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

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

[25] 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.