Snap-Stabilizing Committee Coordination

Borzoo Bonakdarpour, Stéphane Devismes, Franck Petit

Snap-Stabilizing Committee Coordination☆

Borzoo Bonakdarpour

Department of Computing And Software, McMaster University

Stéphane Devismes*

VERIMAG UMR 5104, Université Joseph Fourier, Grenoble

Franck Petit

LIP6 UMR 7606, UPMC Sorbonne Universités, Paris

Abstract

In the *committee coordination problem*, a committee consists of a set of professors and committee meetings are synchronized, so that each professor participates in at most one committee meeting at a time. In this paper, we propose two *snap-stabilizing* distributed algorithms for the committee coordination. *Snap-stabilization* is a versatile property which requires a distributed algorithm to efficiently tolerate transient faults. Indeed, after a finite number of such faults, a snap-stabilizing algorithm immediately operates correctly, without any external intervention. We design snap-stabilizing committee coordination algorithms enriched with some desirable properties related to *concurrency*, *(weak) fairness*, and a stronger synchronization mechanism called *2-Phase Discussion*. In our setting, all processes are identical and each process has a unique identifier. The existing work in the literature has shown that (1) in general, fairness cannot be achieved in committee coordination, and (2) it becomes feasible if each professor waits for meetings infinitely often. Nevertheless, we show that even under this latter assumption, it is impossible to implement a fair solution that allows *maximal concurrency*. Hence, we propose two orthogonal snap-stabilizing algorithms, each satisfying 2-phase discussion, and either maximal concurrency or fairness. The algorithm that implements fairness requires that every professor waits for meetings infinitely often. Moreover, for this algorithm, we introduce and evaluate a new efficiency criterion called the *degree of fair concurrency*. This criterion shows that even if it does not satisfy maximal concurrency, our snap-stabilizing fair algorithm still allows a high level of concurrency.

Keywords: Distributed algorithms, snap-stabilization, self-stabilization, committee coordination

☆A preliminary version of this paper has been published in IPDPS'2011 [1].

*Corresponding author.

Email addresses: `borzoo@mcmaster.ca` (Borzoo Bonakdarpour), `stephane.devismes@imag.fr` (Stéphane Devismes), `franck.petit@lip6.fr` (Franck Petit)

URL: `www.cas.mcmaster.ca/~borzoo/` (Borzoo Bonakdarpour), `www-verimag.imag.fr/~devismes/` (Stéphane Devismes), `http://pagesperso-systeme.lip6.fr/Franck.Petit/` (Franck Petit)

2010 MSC: 68W15, 68M15

1. Introduction

Distributed systems are often constructed based on an asynchrony assumption. This assumption is quite realistic, given the principle that distributed systems must be conveniently expandable in terms of size and geographical scale. It is, nonetheless, inevitable that processes running across a distributed system often need to synchronize for various reasons, such as exclusive access to a shared resource, termination, agreement, rendezvous, *etc.* Implementing synchronization in an asynchronous distributed system has always been a challenge, because of obvious complexity and significant cost; if synchronization is handled in a centralized fashion using traditional shared-memory constructs such as barriers, it may turn into a major bottleneck, and, if it is handled in a fully distributed manner, it may introduce significant communication overhead, unfair behavior, and be vulnerable to numerous types of faults.

The classic *committee coordination problem* [2] characterizes a general type of synchronization called n-ary *rendezvous* as follows:

> "*Professors in a certain university have organized themselves into committees. Each committee has an unchanging membership roster of one or more professors. From time to time a professor may decide to attend a committee meeting; he starts waiting and remains waiting until a meeting of a committee of which he is a member is started. All meetings terminate in finite time. The restrictions on convening a meeting are as follows: (1) meeting of a committee may be started only if all members of that committee are waiting, and (2) no two committees can meet simultaneously, if they have a common member. The problem is to ensure that (3) if all members of a committee are waiting, then a meeting involving some member of this committee is convened.*"

In the context of a distributed system, professors and committees can be mapped onto *processes* and *synchronization events* (*e.g.*, rendezvous) respectively. Moreover, the three properties identified in this definition are known as (1) Synchronization, (2) Exclusion, and (3) Progress, respectively.

Most of the existing algorithms that solve the committee coordination problem [2, 3, 4, 5, 6, 7] overlook properties that are vital in practice. Examples include satisfying fairness or reaching maximum concurrency among convened committees and/or professors in a meeting. Moreover, to our knowledge, none of the existing algorithms is resilient to the occurrence of faults. These features are significantly important when a committee coordination algorithm is implemented to ensure distributed mutual exclusion in code generation frameworks, such as process algebras, *e.g.*, CSP, Ada, and BIP [8].

With this motivation, in this paper, we propose snap-stabilizing [9, 10] distributed algorithms for the committee coordination problem, where all processes are identical and each process has a unique identifier. Snap-stabilization is a versatile property which requires a distributed algorithm to efficiently tolerate transient faults. Indeed, after a finite number of such faults (*e.g.,* memory corruptions, message losses, *etc.*), a snap-stabilizing algorithm immediately operates correctly, without any external (*e.g.,* human) intervention. A snap-stabilizing algorithm is also a *self-stabilizing*

[11] algorithm that stabilizes in 0 steps. In other words, our algorithms are optimal in terms of stabilization time, *i.e.*, every meeting convened *after* the last fault satisfies every requirement of the committee coordination. By contrast, an algorithm that would be only self (but not snap) stabilizing only recovers a correct behavior in finite time after the occurrence of the last fault. Nevertheless, to the best of our knowledge, the committee coordination problem was never addressed in the area of self-stabilization. Therefore, the algorithms proposed in this paper are also the first self-stabilizing committee coordination protocols.

Our snap-stabilizing committee coordination algorithms are enriched with other desirable properties. These properties include Professor Fairness, Maximal Concurrency, and 2-Phase Discussion. The former property means that every professor which requests to participate in a committee meeting that he is a member of, eventually does. Roughly speaking, the second of the aforementioned properties consists in allowing as many committees as possible to meet simultaneously. The latter (2-Phase Discussion) requires professors to collaborate for a minimum amount of time before leaving a meeting.

We first consider Maximal Concurrency and Professor Fairness. As in [7], to circumvent the impossibility of satisfying fairness [5], each time we consider professor fairness in the sequel of the paper, we assume that every professor waits for a meeting infinitely often. Under this assumption, we show that Maximal Concurrency and Professor Fairness are two mutually exclusive properties, *i.e.*, it is impossible to design a committee coordination algorithm (even non-stabilizing) that satisfies both features simultaneously.

Consequently, we focus on the aforementioned contradictory properties independently by providing the two snap-stabilizing algorithms. The former maximizes concurrency at the cost of not ensuring professor fairness. On the contrary, the second algorithm maintains professor fairness, but maximal concurrency cannot be guaranteed. Both algorithms are based on the straightforward idea that coordination of the various meetings must be driven by a priority mechanism that helps each professor to know whether or not he can participate in a meeting. Such a mechanism can be implemented using a token circulating among the professors. To ensure fairness, when a professor holds a token, he has the higher priority to convene a meeting. He then retains the token until he joined the meeting. In that case, some neighbors of the token holder can be prevented from participating in other meetings so that the token holder eventually does. This results in decreasing the level of concurrency. In order to guarantee maximal concurrency (but at the risk of being unfair), a waiting professor must release the token if he is not yet able to convene a meeting to give a chance to other committees in which all members are already waiting.

Thus, in the first algorithm, we show the implementability of committee coordination with Maximal Concurrency even if professors are not required to wait for meetings infinitely often. To the best of our knowledge this is the first committee coordination algorithm that implements maximal concurrency. Moreover, the algorithm is snap-stabilizing and satisfies 2-Phase Discussion.

We also propose a snap-stabilizing algorithm that satisfies Fairness on professors (respectively, committees) and respects 2-Phase Discussion. As mentioned earlier, this algorithm assumes that every professor waits for a meeting infinitely often. Following our impossibility result, the algorithm does not satisfy Maximal Concurrency. However, we show that it still allows a high level of concurrency. We analyze this level of concurrency according to a newly defined criterion called the degree of fair concurrency. We also study the waiting time of our algorithm.

Organization. The rest of the paper is organized as follows. In Section 2, we present the preliminary concepts. Section 3 is dedicated to definitions of Maximal Concurrency and Fairness in committee coordination. Then, in Section 4, we propose our first snap-stabilizing algorithm that satisfies both Maximal Concurrency and 2-phase Discussion. In Section 5, we present our snap-stabilizing algorithm that satisfies Fairness and 2-phase Discussion. Our analysis on level of concurrency and waiting time is also presented in this section. Related work is discussed in Section 6. Finally, we present concluding remarks and discuss future work in Section 7.

2. Background

2.1. Distributed Systems as Hypergraphs

Considering the committee coordination problem in the context of distributed systems, professors and committees are mapped onto *processes* and *synchronization events* (*e.g.*, rendezvous) respectively. We assume that each process has a unique identifier and the set of all identifiers is a total order. We simply denote the identifier of a process p by p.

For the sake of simplicity, we assume that each committee has at least two members.[1] Hence, we model a *distributed system* as a simple self-loopless hypergraph $\mathcal{H} = (V, \mathcal{E})$ where V is a finite set of vertices representing processes and $\mathcal{E}$ is a finite set of *hyperedges* representing synchronization events, such that for all $\epsilon \in \mathcal{E}$, we have $\epsilon \in 2^V$, *i.e.*, each hyperedge is formed by a subset of vertices.

Let v be a vertex in V and ϵ be a hyperedge in $\mathcal{E}$. We denote by $v \in \epsilon$ the fact that vertex v is incident to hyperedge ϵ. We denote the set of hyperedges incident to vertex v by $\mathcal{E}_v$. We say that two distinct vertices u and v are *neighbors* if and only if u and v are incident to some hyperedge ϵ; *i.e.*, there exists $\epsilon \in \mathcal{E}$, such that $u, v \in \epsilon$. The set of all neighbors of v is denoted by $N(v)$.

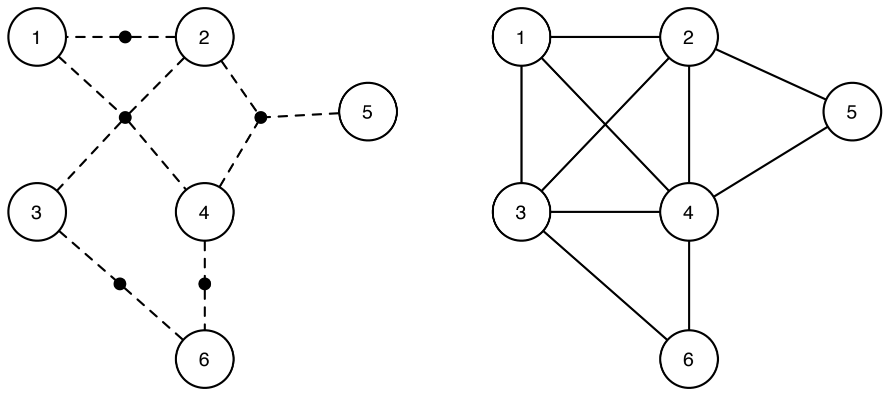
In the committee coordination problem, professors in the same committee need to communicate with each other. We assume that two processes can directly communicate with each other if and only if they are neighbors. This induces what we call an underlying communication network defined as follows: the *underlying communication network* of a distributed system $\mathcal{H} = (V, \mathcal{E})$ is an undirected simple connected graph $G_{\mathcal{H}} = (V, E_{\mathcal{E}})$, where $E_{\mathcal{E}} = \{\{p_1, p_2\} \mid p_1 \in V \wedge p_2 \in V \wedge p_1 \in N(p_2)\}$. Figure 1(b) shows the underlying communication network of the hypergraph given in Figure 1(a).

2.2. Computational Model

The communication between processes are carried out using *locally shared variables*. Each process owns a set of locally shared variables, henceforth referred to as *variables*. Each variable ranges over a fixed domain and the process can read and write them. Moreover, a process can also read variables of its neighbors.[2] The *state* of a process is defined by the value of its variables. A

[1] Adapting our results to take singleton committees into account is straightforward.

[2] In particular, a process can read the identifiers of its neighbors.



(a) Hypergraph $\mathcal{H} = (V, \mathcal{E})$, where $V = \{1, 2, 3, 4, 5, 6\}$ and $\mathcal{E} = \{\{1, 2\}, \{1, 2, 3, 4\}, \{2, 4, 5\}, \{3, 6\}, \{4, 6\}\}$.

(b) Graph $G_{\mathcal{H}} = (V, E_{\mathcal{E}})$, where $E_{\mathcal{E}} = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 4\}, \{3, 6\}, \{4, 5\}, \{4, 6\}\}$

Figure 1: An example of a hypergraph and its underlying communication network.

process can change its state by executing its *local algorithm*. The local algorithm of a process p is described using a finite *ordered list* of *guarded actions* of the form:

$$\langle label \rangle \; :: \langle guard \rangle \; \mapsto \; \langle statement \rangle.$$

The *label* of an action is only used to identify the action in discussions and proofs. The *guard* of an action of p is a Boolean expression involving a subset of variables of p and its neighbors. The *statement* of an action of p updates a subset of variables of p. The order of the list follows the order of appearance of the actions in the code of the local algorithm and give priorities to actions: action A has higher priority than action B if and only if A appears after B in the code.

A *configuration* γ in a distributed system is an instance of the state of its processes. We denote the set of all configurations of a distributed system $\mathcal{H}$ by $\Gamma_{\mathcal{H}}$. The concurrent execution of the set of all local algorithms defines a *distributed algorithm*. We say that an action of a process p is *enabled* in a configuration γ if and only if its guard is true in γ. By extension, process p is said to be enabled in γ if and only if at least one of its actions is enabled in γ. An action can be executed only if its guard is enabled. We denote by *Enabled*(γ) the subset of processes that are enabled in configuration γ.

When the configuration is γ and *Enabled*$(\gamma) \neq \emptyset$, a *daemon* (or *scheduler*) selects a non-empty set $\mathcal{X} \subseteq$ *Enabled*(γ); then every process of $\mathcal{X}$ *atomically* executes its priority enabled action, leading to a new configuration γ', and so on. The transition from γ to γ' is called a *step* (of $\mathcal{A}$). The possible steps induce a binary relation over configurations of $\mathcal{A}$, denoted by $\mapsto$.

A *computation* of a distributed system is a maximal sequence of configurations $\gamma_0, \gamma_1, \ldots$ such that (1) γ_0 is an arbitrary configuration, and (2) for each configuration γ_i, with $i \geq 0$, $\gamma_i \mapsto \gamma_{i+1}$. *Maximality* of a computation means that the computation is either infinite or eventually reaches a terminal configuration (*i.e.*, a configuration where no action is enabled).

A daemon is defined as a predicate over computations. There exist several kinds of daemons. Here, we consider a *distributed weakly fair* daemon. *Distributed* means that, at each step, if one or more processes are enabled, then the daemon selects at least one (maybe more) of these processes. *Weak fairness* means that every continuously enabled process is eventually selected by the daemon.

We say that a process p is *neutralized* in $\gamma_i \mapsto \gamma_{i+1}$, if p is *enabled* in γ_i and not enabled in γ_{i+1}, but did not execute any action in $\gamma_i \mapsto \gamma_{i+1}$. To compute the time complexity, we use the notion of *round* [12]. This notion captures the execution rate of the slowest process in any computation. The first *round* of a computation e is the minimal prefix of e, $\gamma_0 \dots \gamma_i$, containing the activation or the neutralization of every process that is enabled in the initial configuration. Let e_{γ_i} be the suffix of e starting from γ_i (the last configuration of the first round of e). The second *round* of e is the first round of e_{γ_i}, and so on.

The *fair composition* [13] of two algorithms $\mathcal{P}_1$ and $\mathcal{P}_2$ consists in running $\mathcal{P}_1$ and $\mathcal{P}_2$ in alternation in such a way that there is no computation suffix, where a process is continuously enabled *w.r.t.* $\mathcal{P}_i$ ($i \in \{1, 2\}$) without executing any of its enabled actions *w.r.t.* $\mathcal{P}_i$.

2.3. The Committee Coordination Problem

The *original committee coordination problem* is as follows [2]. Let $\mathcal{H} = (V, \mathcal{E})$ be a distributed system. Each process in V represents a *professor* and each hyperedge in $\mathcal{E}$ represents a committee. We say that two committees ϵ_1 and ϵ_2 are *conflicting* if and only if $\epsilon_1 \cap \epsilon_2 \neq \emptyset$. A professor can be in anyone of the following three states: (1) *idle*, (2) *waiting*, and (3) *meeting*. A professor may remain in the idle state for an arbitrary (even infinite) period of time. An idle professor may start waiting for a committee meeting. A professor remains waiting until all participating professors of a committee, which he is a member of, agree on meeting. Moreover, a professor may leave a meeting, become idle, and subsequently be waiting for a new committee meeting.

Chandy, Misra [2], and Bagrodia [4] require that any solution to the problem must satisfy the following specification:

- *(Exclusion)* No two conflicting committees may meet simultaneously.
- *(Synchronization)* A committee meeting may convene only if all members of that committee are waiting.
- *(Progress)* If all members of a committee ϵ are waiting, then some professor in ϵ eventually goes to the meeting state.

2.4. 2-Phase Discussion

The original Committee Coordination problem specification does not constrain professors with respect to their time spent in a committee meeting in any ways. Thus, distributed algorithms for committee coordination have been developed regardless this issue. For instance, solutions proposed in [2, 4] that employ the dining philosophers problem [14] in order to resolve committee conflicts satisfy the specification presented in Subsection 2.3, but have the following shortcoming. Since a philosopher acquires and releases forks all at once, members of the corresponding committee have to leave the meeting all together.[3] There are two problems with such a restriction: (1)

[3]The same argument holds for solutions based on the *drinking philosophers* [14] and tokens.

an implicit strong synchronization is assumed on terminating a committee meeting, and (2) fast professors have to wait for slow professors to finish the task for which they setup a rendezvous.

We constrain the specification such that upon agreement on a meeting, the meeting takes place until a professor unilaterally leaves (that is, without waiting for other professors) the meeting. The reason for this requirement is due to the fact that in practical settings, based upon the speed of processes (professors), the type of local computation, and required resources, each process may spend a different time period to utilize resources or execute a critical section. Nevertheless, we also require that each professor must spend a minimum amount of time to discuss issues in the meeting. The intuition for this constraint is that processes participate in a rendezvous to share resources or do some minimal computation and, hence, they should not be allowed to leave the meeting immediately after it convenes. Another reason for requiring this minimal discussion by all professors is inspired by the fact that in the recent applications of using rendezvous interactions to generate correct distributed and multi-core code, such interactions normally involve data transmission and even code execution at interaction level [15, 16]. The following definition elegantly captures this requirement.

Definition 1 (2-Phase Discussion) *We define the 2-phase discussion by the following two properties:*

- `Phase 1.` *(Essential Discussion) Upon a meeting convenes, a first session of discussion should take place until each participating professor has the opportunity to execute a task involving information from all or part of the participants.*

- `Phase 2.` *(Voluntary Discussion) Upon a meeting convenes and after fulfilling the essential discussion, the discussion (and consequently the meeting) continues until a professor voluntarily terminates his/her discussion (and consequently the meeting).*

In the following, we call *2-phase committee coordination problem* the committee coordination problem enriched with the essential and voluntary discussions.

2.5. Snap-stabilization

Snap-stabilization [9, 10] is a versatile property which requires a distributed algorithm to efficiently tolerate transient faults. Indeed, after a *finite number* of such faults (*e.g.*, memory corruptions), a snap-stabilizing algorithm *immediately* operates correctly, without any external (*e.g.* human) intervention. By contrast, the related concept of self-stabilization [11] only guarantees that the system *eventually* recovers to a correct behavior.

In (self- or snap-) stabilizing systems, we consider the system immediately after the occurrence of the *last* fault. That is, we study the system starting from an arbitrary configuration reached due to the occurrence of transient faults, but from which *no fault will ever occur*. By abuse of language, this configuration is referred to as initial configuration of the system in the literature. A snap-stabilizing algorithm then guarantees that *starting from any arbitrary initial configuration, any of its computations always satisfies the specification of the problem.*

This means, in particular, that in (self- or snap-) stabilizing systems there is no fault model in the literal sense. As we study the system after the *last* fault, we do not treat the faults but their consequences. The result of a finite number of transient faults being the arbitrary perturbation of the system configuration, we consider any computation started in any arbitrary initialized configuration, but in which there is no fault. So, for example, to show that our algorithms are snap-stabilizing *w.r.t* the committee coordination problem, we have to show that the specification of the committee coordination problem (*e.g.*, exclusion, progress, synchronization, *etc*) is always satisfied in *all* possible (fault-free) computations starting from all possible (arbitrary) configurations.

It is important to note that snap-stabilizing algorithms are not insensitive to transient faults. Actually, a snap-stabilizing algorithm guarantees that any task execution started *after* the end of the faults operates correctly. However, there is no guarantees for tasks executed completely or in part during faults. By contrast, self- but not snap- stabilizing algorithms require to start task execution several times (yet a finite number of time) before correctly performing them (that is, *w.r.t.* their specification). Hence, snap-stabilization is a specialization of self-stabilization that offers stronger safety guarantees. For example, in the committee coordination problem, snap-stabilization ensures that every meeting convened after the last transient faults satisfies every requirement of the committee coordination problem. However, there is no guarantees for the meetings started during the transient faults, except that they do not interfere with the execution of the meetings that convened after the last fault.

3. Maximal Concurrency versus Fairness in Committee Coordination

3.1. Definitions

In practical applications, it is crucial to allow as many processes as possible to execute simultaneously without violating other correctness constraints. Although the level of concurrency has significant impact on performance and resource utilization, it does not appear as a constraint in the original committee coordination problem. Moreover, the solutions proposed by Chandy and Misra [2] and Bagrodia [3, 4] result in decreasing the level of concurrency drastically, making them less appealing for practical purposes. Examples include the circulating token mechanism among conflicting committees [3], and reduction to the dining philosophers problems, where a "*manager*" handles multiple committees. Reduction to the drinking philosophers problem such as those in [2, 4, 17] results in *more* concurrency, but not maximal. This is due to the fact that existing solutions to the drinking philosophers problem try to achieve concurrency and fairness simultaneously, which we will show is impossible in committee coordination.

We formulate the issue of concurrency, so that as many committees as possible meet simultaneously. Our definition of maximal concurrency is inspired by the efficiency property given in [18]. Informally, we define maximal concurrency as follows: if there is at least one committee, such that all its members are waiting, then eventually a new meeting convenes *even if no other meeting terminates in the meantime*. In other words, while it is possible, new meetings should be able to convene, regardless the duration of meetings that already hold. Now, to formally define maximal concurrency we need, in particular, to express the constraint "regardless of the duration of meetings that already hold". For that purpose, we borrow the ideas of Datta *et al* [18] by using the following artefact: we let a professor (process) remains in the meeting state forever. We emphasize that we make this assumption only to define our constraint; our results in this paper do assume finite-time meetings as mentioned earlier.

Definition 2 (Maximal Concurrency) *Assume that there is a set of professors P_1 that are all in infinite-time meetings. Let P_2 be a set of professors waiting to enter a committee meeting (Obviously, $P_1 \cap P_2 = \emptyset$ and idle processes are in neither P_1 nor P_2). Let Π be the set of hyperedges having all their incident professors in P_2. If $\Pi \neq \emptyset$, then a meeting between every professor incident to some hyperedge $\epsilon \in \Pi$ eventually convenes.*

We note that in Definition 2, we use the term "maximal", because our intention is not to enforce the largest number of committees (*i.e.*, maximum) to meet simultaneously, this latter problem is clearly $\mathcal{NP}$-hard! In other words, committees convene until the systems is exhausted. This greedy approach does not always result in obtaining the maximum number of committees that can meet at the same time.

Following the results in [5], if a professor's status does not become waiting infinitely often, achieving fairness is impossible. Thus, we consider fairness assuming professors always eventually switch to the waiting status. In this context, we define fairness on professors (also called weak fairness, [6]) as follows.

Definition 3 (Professor Fairness) *Every professor participates infinitely often in a committee meeting that he is a member of.*

3.2. Negative Result

The next theorem shows that Maximal Concurrency and Professor Fairness are incompatible. Its proof follows ideas similar to the impossibility results of Joung [19] as well as Tsay and Bagrodia [5].

The idea behind this result is rather simple: Consider any process p. To satisfy professor fairness, a meeting having p as member must eventually convene. To have such a guarantee, the algorithm may eventually have to prevent some neighbors of p from participating in meetings until a meeting including them and p can convene. These blockings may happen while no meeting including p can be yet convened. This constraint then prevents some meetings from holding concurrently. That is, making maximal concurrency impossible.

Theorem 1 *Assuming that every professor waits for meetings infinitely often, it is impossible to design an algorithm (even non-stabilizing) for an arbitrary distributed system that solves the committee coordination problem and simultaneously satisfies Maximal Concurrency and Professor Fairness.*

Proof. Suppose by contradiction that there exists an algorithm $\mathcal{A}$ (be it stabilizing or not) working in any topology that satisfies both Maximal Concurrency and Professor Fairness. Now, consider a computation of $\mathcal{A}$ on hypergraph $\mathcal{H} = (V, \mathcal{E})$ where $V = \{1, 2, 3, 4, 5\}$ and $\mathcal{E} = \{\{1, 2\}, \{1, 3, 5\}, \{3, 4\}\})$. Figure 2 shows three possible configurations A, B, and C obtained by executing algorithm $\mathcal{A}$ on $\mathcal{H}$. In the figure, solid bold lines represent meetings that are currently being held. Also, a process that is not in a meeting is supposed to be waiting. For example, in configuration A, professors 1 and 2 are meeting and professors 3, 4, and 5 are waiting.

We first show that there are computations of $\mathcal{A}$ that eventually reach configuration A. As professors 1 and 2 wait for meetings infinitely often, by Professor Fairness, a meeting between

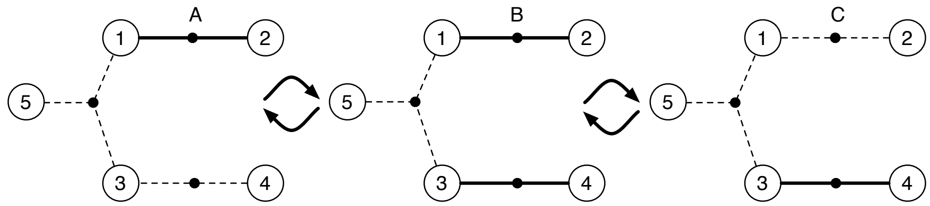


Figure 2: Impossibility of Maximal Concurrency and Professor Fairness.

professors 1 and 2 eventually convenes. When this happens, if professors 3 and 4 are meeting, then their meeting can terminate before the one between 1 and 2. So, the system may reach a configuration where only 1 and 2 are meeting. After that, assuming that professors 3, 4, and 5 immediately go to the waiting state, then the system reaches configuration A.

From configuration A, if the committee $\{1, 2\}$ takes an arbitrary long (but finite) time, then a meeting of the committee $\{3, 4\}$ must eventually convene in order to satisfy Maximal Concurrency and the system reaches configuration B. Now, suppose meeting $\{1, 2\}$ terminates first and professors 1 and 2 immediately go to waiting state again. So, 1, 2, and 5 are waiting and 3 and 4 are in a meeting (configuration C). Following a similar reasoning, configuration B can be reached from configuration C, and configuration A can be reached from configuration B. By repeating this pattern infinitely many times, we obtain a possible computation of $\mathcal{A}$, where professor 5 never participates in any meeting while being continuously waiting, which contradicts with Professor Fairness. $\square$

Note that Maximal Concurrency and Professor Fairness can be simultaneously achieved in some particular networks, *e.g.*, networks where no committees are in conflict, or networks where some professor belongs to all committees (*e.g.*, a complete hypergraph, or a star topology). In the latter case, note that all committees are conflicting and so at most one can meet at a time.

We note that every algorithm that satisfies Professor Fairness also satisfies Progress. Also, observe that Professor Fairness does not imply that particular committees eventually convene. We define such a property as follows.

Definition 4 (Committee Fairness) *Every committee meeting convenes infinitely often.*

Notice that since Committee Fairness implies Professor Fairness, impossibility of satisfying both Maximal Concurrency and Committee Fairness trivially follows.

Corollary 1 *Assuming that every professor waits for meetings infinitely often, it is impossible to design an algorithm (even non-stabilizing) for an arbitrary distributed system that solves the committee coordination problem and simultaneously satisfies Maximal Concurrency and Committee Fairness.*

Theorem 1 shows that Professor Fairness and Maximal Concurrency are contradictory properties to satisfy. Thus, in order to satisfy one property, we have to omit the other. Omitting fairness results in an algorithm such as the one presented in Section 4. Omitting maximal concurrency results in an algorithm such as the one presented in Section 5.

Note that both algorithms use a single token circulation that ensures the progress in the former case and the fairness in the latter. As a matter of fact, they mainly differ in the way they handle the token. Concerning the second algorithm, one can suggest that the use of several tokens (*e.g.*, the local mutual exclusion mechanism in [20]) instead of a single one would enhance the fairness guarantee. However, increasing the number of tokens results in decreasing the degree of (fair) concurrency,[4] which is the target metric here. The key idea is that the token is used to give priority to convene a meeting. However, the token is not mandatory to join a meeting, unless a process is starved to join a meeting. Then, to guarantee fairness, it is mandatory that the token holder selects a committee and sticks with that committee until it meets, even if some members of that committee are currently participating in another meeting. In this case, every other waiting member of that committee has to wait until the meeting convenes while they may participate in a meeting of another committee. This results in decreasing the degree of concurrency (that is why our second algorithm does not satisfy Maximal Concurrency): every waiting member of the committee selected by the token holder is blocked until the committee is able to convene. Hence, increasing the number of tokens increases the number of blocked processes which in turn decreases the degree of concurrency. In other word, enforcing the fairness decreases concurrency.

3.3. Complexity Analysis of Fair Solutions

We now introduce and study two complexity measures: *degree of fair concurrency* and *waiting time*. First, in order to characterize the impact of fairness on reducing the number of processes that can run concurrently, we introduce the notion of Degree of Fair Concurrency. Roughly speaking, this degree is the minimum number of committees that can meet concurrently without compromising Professor Fairness.

Definition 5 (Degree of Fair Concurrency) *Let $\mathcal{A}$ be a committee coordination algorithm that satisfies Professor Fairness. Let professors remain in a meeting for infinite time.*[5] *Under such an assumption the system reaches a* quiescent *state where the status of all professors do not change any more. The Degree of Fair Concurrency of $\mathcal{A}$ is then the minimum number of meetings held in a quiescent state.*

When considering fair solutions, it is of practical interest to evaluate the Waiting Time. In our context where processes are either waiting or meeting, we define waiting time as follows:

Definition 6 (Waiting Time) *The maximum time before a process participates in a committee meeting is* waiting time.

4. Snap-stabilizing 2-Phase Committee Coordination with Maximal Concurrency

In this section, we propose a Snap-stabilizing algorithm that satisfies Maximal Concurrency as well as the 2-Phase Discussion. We present our algorithm in Subsection 4.1. The correctness proof appears in Subsection 4.2.

[4] The term "degree of fair concurrency" is formally explained in Subsection 3.3

[5] As in Definition 2, infinite meetings are used only for formalization.

4.1. Algorithm

Our algorithm is a composition of two modules: (1) a Snap-stabilizing algorithm – denoted $\mathcal{CC}1$ – that ensures Exclusion, Synchronization, Maximal Concurrency, and 2-Phase Discussion, and (2) a self-stabilizing module – denoted $\mathcal{TC}$ – that manages a circulating token for ensuring Progress. Each process p runs this algorithm, where the intention of p in participating or leaving a committee are declared by truthfulness of input predicates $RequestIn(p)$ and $RequestOut(p)$, respectively.

Remark 1 *We emphasize that this composition is snap-stabilizing, as the self-stabilizing token circulation is not used to ensure any safety property.*

Token Circulation Module. We assume that the token circulation module is a black box with the following property:

Property 1

- *$\mathcal{TC}$ contains one action to pass the token from neighbor to neighbor:*

$$T :: Token(p) \quad \mapsto \quad ReleaseToken_p$$

- *Once stabilized, every process executes action T infinitely often, but when T is enabled in a process, it is not enabled in any other process.*
- *$\mathcal{TC}$ stabilizes independently of the activations of action T.*

To obtain such a token circulation, one can compose a self-stabilizing leader election algorithm (*e.g.*, in [21, 22, 23]) with one of the self-stabilizing token circulation algorithms in [24, 25, 26, 27] for arbitrary rooted networks. The composition only consists of two algorithms running concurrently with the following rule: if a process decides that it is the leader, it executes the root code of the token circulation. Otherwise, it executes the code of the non-root process.

Composition. The composition of $\mathcal{CC}1$ and $\mathcal{TC}$ is denoted by $\mathcal{CC}1 \circ \mathcal{TC}$. Actually, $\mathcal{CC}1 \circ \mathcal{TC}$ is a fair composition of $\mathcal{CC}1$ and $\mathcal{TC}$ that does not explicitly contain action T: in $\mathcal{CC}1 \circ \mathcal{TC}$, action T is emulated by $\mathcal{CC}1$, where predicate $Token(p)$ and the statement $ReleaseToken_p$ are given as inputs in $\mathcal{CC}1$.

Committee Coordination Module. Algorithm $\mathcal{CC}1$ is identical for all processes in the distributed system. Its code is given in Algorithm 1. Interactions between each professor p and his local algorithm are managed using two input predicates: $RequestIn(p)$ and $RequestOut(p)$. These predicates express the fact that a professor autonomously decides to wait and leave a meeting, respectively. The predicate $RequestIn(p)$ holds when professor p requests participation in a committee meeting. The predicate $RequestOut(p)$ holds when p desires to stop discussing in a meeting. Thus, p eventually satisfies $RequestOut(p)$ during the meeting or after some members left it. So, once p has done its essential discussion, it can voluntary leave the meeting when it satisfies $RequestOut(p)$.

Each process p maintains a status variable $S_p \in \{\mathsf{idle}, \mathsf{looking}, \mathsf{waiting}, \mathsf{done}\}$, a Boolean variable T_p, and an edge pointer P_p. We explain the goal of these variables below:

Algorithm 1 Pseudo-code of $\mathcal{CC}1$ for process p.

Inputs:

$RequestIn(p)$	:	Predicate: input from the system indicating desire for participating in a committee
$RequestOut(p)$	:	Predicate: input from the system indicating desire for leaving a committee
$Token(p)$	:	Predicate: input from $\mathcal{TC}$ indicating process p owns the token
$ReleaseToken(p)$	:	Statement: output to $\mathcal{TC}$ indicating process p releases the token

Constants:

$\mathcal{E}_p$: Set of hyperedges incident to process p

Variables:

$S_p \in \{\mathsf{idle}, \mathsf{looking}, \mathsf{waiting}, \mathsf{done}\}$	:	Status
$P_p \in \mathcal{E}_p \cup \{\perp\}$	:	Edge pointer
T_p	:	Boolean

Macros:

$$
\begin{array}{lcl}
FreeEdges_p & = & \{\epsilon \in \mathcal{E}_p \mid \forall q \in \epsilon \ : \ S_q = \mathsf{looking}\} \\
FreeNodes_p & = & \{q \mid \exists \epsilon \in FreeEdges_p \ : \ q \in \epsilon\} \\
TFreeNodes_p & = & \{q \in FreeNodes_p \mid T_q\} \\
Cands_p & = & \textbf{if } (TFreeNodes_p \neq \emptyset) \textbf{ then } TFreeNodes_p \textbf{ else } FreeNodes_p \textbf{ fi}
\end{array}
$$

Predicates:

$$
\begin{array}{lcl}
Ready(p) & \equiv & \exists \epsilon \in \mathcal{E}_p \ : \ \forall q \in \epsilon \ : \ ((P_q = \epsilon) \ \wedge \ (S_q \in \{\mathsf{looking}, \mathsf{waiting}\})) \\
LocalMax(p) & \equiv & p = \max(Cands_p) \\
MaxToFreeEdge(p) & \equiv & (FreeEdges_p \neq \emptyset) \ \wedge \ LocalMax(p) \ \wedge \ \neg Ready(p) \ \wedge \ (P_p \notin FreeEdges_p) \\
JoinLocalMax(p) & \equiv & (FreeEdges_p \neq \emptyset) \ \wedge \ \neg LocalMax(p) \ \wedge \ \neg Ready(p) \ \wedge \\
 & & (\exists \epsilon \in FreeEdges_p \ : \ (P_{\max(Cands_p)} = \epsilon \ \wedge \ P_p \neq \epsilon)) \\
Meeting(p) & \equiv & \exists \epsilon \in \mathcal{E}_p \ : \ \forall q \in \epsilon \ : \ (P_q = \epsilon \ \wedge \ S_q \in \{\mathsf{waiting}, \mathsf{done}\}) \\
LeaveMeeting(p) & \equiv & \exists \epsilon \in \mathcal{E}_p \ : \ ((P_p = \epsilon) \ \wedge \ (\forall q \in \epsilon \ : \ ((P_q = \epsilon) \ \Rightarrow \ (S_q = \mathsf{done})))) \\
Useless(p) & \equiv & Token(p) \ \wedge \ [(S_p = \mathsf{idle}) \ \vee \ (S_p = \mathsf{looking} \ \wedge \ FreeEdges_p = \emptyset)] \\
Correct(p) & \equiv & [(S_p = \mathsf{idle}) \quad \Rightarrow \ (P_p = \perp)] \ \wedge \\
 & & [(S_p = \mathsf{waiting}) \ \Rightarrow \ Ready(p) \ \vee \ Meeting(p)] \wedge \\
 & & [(S_p = \mathsf{done}) \quad \Rightarrow \ Meeting(p) \ \vee LeaveMeeting(p)]
\end{array}
$$

Actions:

$$
\begin{array}{llll}
Step_1 & :: & RequestIn(p) \wedge (S_p = \mathsf{idle}) & \mapsto \ S_p := \mathsf{looking}; P_p := \perp; \\
 & & & \\
Step_{21} & :: & MaxToFreeEdge(p) & \mapsto \ P_p := \epsilon, \text{ such that } \epsilon \in FreeEdges_p; \\
Step_{22} & :: & JoinLocalMax(p) & \mapsto \ P_p := \epsilon, \text{ such that } (\epsilon \in \mathcal{E}_p \ \wedge \ \epsilon = P_{\max(Cands_p)}); \\
 & & & \\
Token_1 & :: & Token(p) \neq T_p & \mapsto \ T_p := Token(p); \\
Token_2 & :: & Useless(p) & \mapsto \ ReleaseToken(p); T_p := false; \\
 & & & \\
Step_{31} & :: & Ready(p) \wedge (S_p = \mathsf{looking}) & \mapsto \ S_p := \mathsf{waiting}; \\
Step_{32} & :: & Meeting(p) \wedge (S_p = \mathsf{waiting}) & \mapsto \ \langle \mathsf{EssentialDiscussion} \rangle; S_p := \mathsf{done}; \\
 & & & \\
Step_4 & :: & LeaveMeeting(p) \wedge RequestOut(p) & \mapsto \ S_p := \mathsf{idle}; P_p := \perp; \textbf{if } Token(p) \textbf{ then } ReleaseToken(p) \textbf{ fi}; T_p := false; \\
 & & & \\
Stab_1 & :: & \neg Correct(p) \ \wedge \ (S_p = \mathsf{idle}) & \mapsto \ P_p := \perp; \\
Stab_2 & :: & \neg Correct(p) \ \wedge \ (S_p \neq \mathsf{idle}) & \mapsto \ S_p := \mathsf{looking}; P_p := \perp;
\end{array}
$$

1. When process p is idle (that is $S_p = \mathsf{idle}$) but desires to participate in a committee meeting (that is, if $RequestIn(p)$ is $true$), it changes its status from idle to looking and initializes its edge pointer P_p to $\perp$ (action $Step_1$).
2. Next, process p starts looking for an available committee to join. Process p shows interest in joining a committee whose processes are all looking by setting its edge pointer P_p to the corresponding hyperedge, if such a hyperedge exists (actions $Step_{21}$ and $Step_{22}$).
   To obtain agreement on the committees to convene, we implement token-based priorities. When a looking process p is the one with highest priority in its neighborhood, it points to an edge corresponding to a committee whose processes are all looking (if any) and sticks with it. Looking processes with low priorities select the committee chosen by their looking neighbor of highest priority, described next.
   Each process p maintains a Boolean variable T_p which shows whether or not it owns a token. A token holder has a higher priority than its neighbors to convene a committee. In case of several token holders (only during the stabilization of token circulation), we give priority to the looking token holder with the maximum identifier.
   A token holder releases its token in two cases: (1) when it leaves a meeting or (2) when it is currently not guaranteed to eventually convene a committee (that is, in each of its incident committees, at least one member is not looking). Note that the algorithm does not guarantee fairness because of this latter case.
   In order to guarantee Maximal Concurrency, we have to authorize committees to meet when all members are looking and if there is no looking token holder in the neighborhood. In this case, among the looking processes we give priority to the looking process with the maximum identifier.
3. Once all processes of a hyperedge are looking and agree on that hyperedge, they are all ready to start their discussion. To this end, a process changes its status from looking to waiting[6] to show that it is waiting for the committee to convene (action $Step_{31}$). A meeting of the committee convenes when all its members change their status to waiting. Then, each process executes its essential discussion and then switches its status to done (action $Step_{32}$).
4. Finally, a process is allowed to leave the committee meeting when all processes of that committee have fulfilled their essential discussion, *i.e.*, they are all in the done status. In this case, the meeting takes place until a process p unilaterally decides to leave it (that is, until $RequestOut(p)$ is $true$) after a finite period of voluntary discussion. To leave the committee meeting, it switches its status to idle again, resets its hyperedge pointer, and releases the token if it owns it (action $Step_4$). Then, the committee meeting is terminated, and every other member q switches to idle since it satisfies $RequestOut(q)$.

The rest of actions of the algorithm deal with token circulation and snap-stabilization. In particular, action $Token_1$ deals with setting variable T_p to $true$, so that neighboring processes realize that p owns the token. If p owns the token and has no desire to take part in a committee meeting, or, there does not exist an available committee for p to participate, then it releases the

[6]Note that both looking and waiting status form the waiting state of the original problem specification [2].

token (action $Token_2$). Finally, actions $Stab_1$ and $Stab_2$ correct the state of a process, if faults perturb the state of the process to a state where predicate $Correct$ does not hold. Predicate $Correct$ holds at states where (1) the process is idle and it has no interest in participating in a committee meeting, (2) it is waiting and interested in a committee whose processes are gathering to convene a meeting, and (3) it has fulfilled its essential discussion and other processes in the corresponding committee are either in $\{$waiting, done$\}$ status, or, the meeting is terminated, that is some processes have left the meeting and the others are done in the meeting.

Example. In this paragraph, we illustrate the need of the token to ensure progress. Figure 3 provides an example of computation that starts from a configuration where each professor state is correct. In the figure, each circle represents a professor and arrows inside the circle represent the P-pointers (if a circle contains no arrow, this means that the corresponding professor p satisfies $P_p = \perp$). Numbers represent identifiers. The status of the professors is given below the circles. The token holder is represented by a bold circle. A boxed "T" near a circle means that the corresponding professor p satisfies $T_p = true$.

In this example, professors in the committee $\{5, 6\}$ desire to participate in a meeting. So, at least one of them should eventually does, according to the progress property. Because they have low identifiers, we can prevent them from convening a meeting until at least one of them get the token.

In 3(a), two meetings are almost done: $\{9, 10\}$ and $\{1, 2, 3\}$, that is, all involved professors are doing their voluntary discussion. Notice that Professor 1 holds the token and $T_1 = true$. Professor 4 is currently not interesting in convening any meeting. All other professors are looking for convening a meeting and point to their highest priority all-looking committee. Now, Professors 7 and 8 are agreeing to convene a meeting: they are both enabled to switch to the waiting status.

In Step 3(a)$\mapsto$3(b), all members of meetings $\{1, 2, 3\}$ and $\{9, 10\}$ simultaneously leave the meeting by executing $Step_4$. Moreover, Professor 8 switches to the waiting status by executing $Step_{31}$. Note in particular that Professor 1 releases the token and resets T_1 to false. Professor 2 is now the token holder. Since his status is idle, he is enabled to release the token. Professor 2 will release the token without setting T_2 to $true$ in the meantime.

In Step 3(b)$\mapsto$3(c), Professor 7 switches to status waiting. So, the meeting $\{7, 8\}$ convenes. In the meantime, both Professors 9 and 10 start again to look for a meeting by executing $Step_1$. Moreover, Professor 2 releases the token. So, in configuration 3(c), Professor 3 is the token holder and Professor 6 should look for another meeting. For Professor 6, the committee of highest priority is $\{6, 9\}$. Similarly, Professor 9 (resp. Professor 10) considers $\{9, 10\}$ as the one of highest priority.

In Step 3(c)$\mapsto$3(d), Professor 3 releases the token, Professors 7 and 8 perform their essential discussion ($Step_{32}$), Professors 10 ($Step_{21}$) and 9 ($Step_{22}$) agree to convene a meeting, and Professor 6 points to Committee $\{6, 9\}$. Note that Professor 4 is the token holder in configuration 3(d), but he has no interest in convening any meeting so his action $Token_2$ is enabled.

In Step 3(d)$\mapsto$3(e), Professor 4 releases the token, Professors 8 and 9 leave their meeting ($Step_4$), and Professor 10 switches to the waiting status by executing $Step_{31}$. In configuration 3(e), Professor 6 is the token holder, consequently he has highest priority. However, meeting $\{8, 9\}$ is ready to convene, so Professor 9, in particular, will not change his pointer P_9.

In Step 3(e)$\mapsto$3(f), Professor 9 switches to status waiting, so the meeting of Committee $\{9, 10\}$

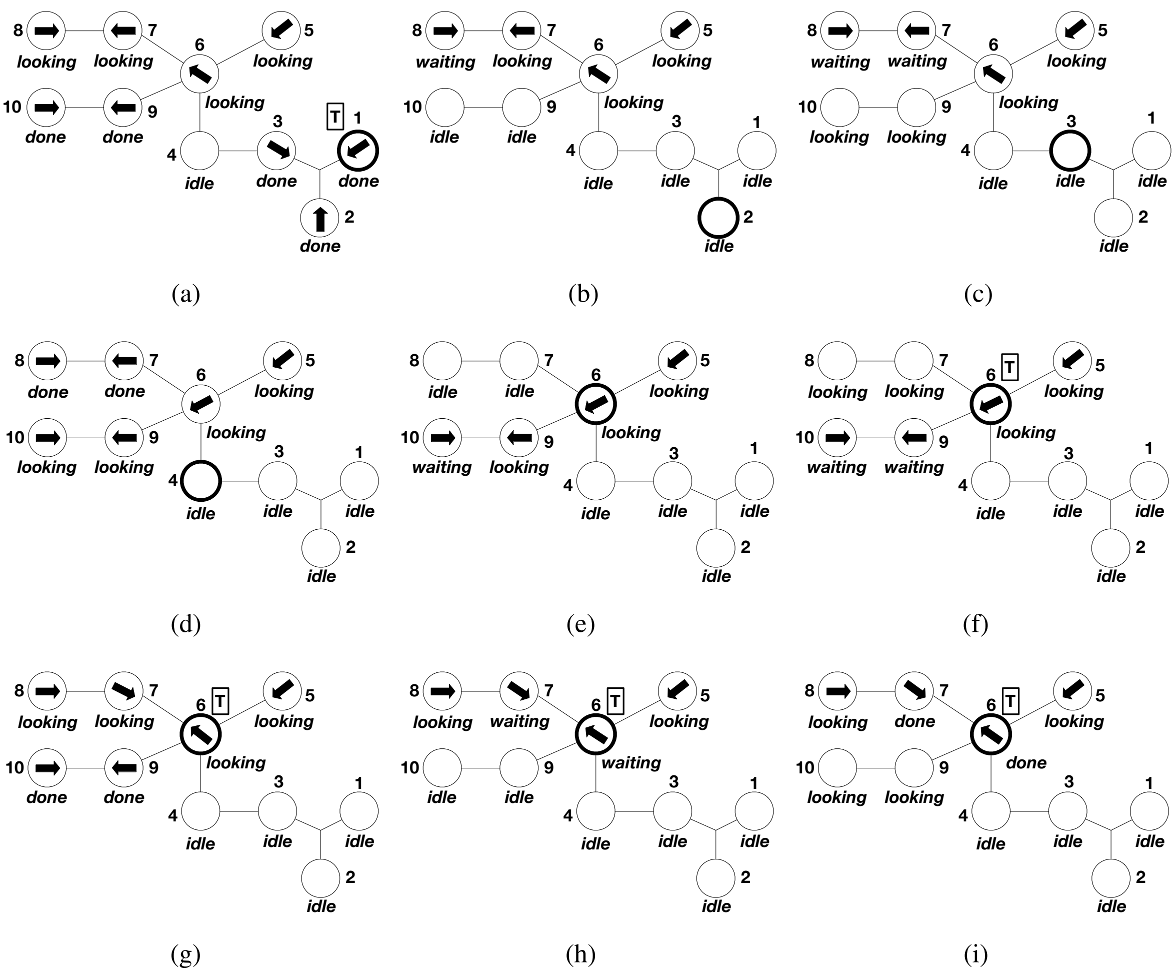


Figure 3: Example

convenes. In the meantime, Professors 8 and 9 start again to look for a meeting by executing $Step_1$. Finally, Professor 6 executes $T_6 \leftarrow true$ ($Token_1$) to inform all its neighbors that he is the token holder. In configuration 3(f), Professors 5, 6, 7, and 8 are all looking for a meeting like in configuration 3(a), but this time Committee $\{6, 7\}$ has the highest priority.

In Step 3(f)$\mapsto$3(g), Professors 9 and 10 perform their essential discussion ($Step_{32}$) and Professors 6 ($Step_{21}$) and 7 ($Step_{22}$) agree to convene a meeting (Professor 8 also executes $Step_{22}$).

In Step 3(g)$\mapsto$3(h), the meeting of Committee $\{9, 10\}$ ends because Professors 9 and 10 simultaneously leave it, and a meeting of Committee $\{6, 7\}$ convenes because Processors 6 and 7 both execute $Step_{31}$.

In Step 3(h)$\mapsto$3(i), Professors 6 and 7 perform their essential discussion ($Step_{32}$). Moreover, Professors 10 and 9 start again to look for a meeting by executing $Step_1$.

4.2. Correctness of Algorithm $\mathcal{CC}1 \circ \mathcal{TC}$

We recall that in the following proofs, we assume that computations of $\mathcal{CC}1 \circ \mathcal{TC}$ start from arbitrary configurations. First, we define the terminology used in the proofs.

We map the state of a professor defined in Section 2.3 to the status of a process defined in Algorithm 1 as follows. We say that a process p is *idle* if and only if $S_p = \mathsf{idle}$. A process p is *waiting* if and only if $S_p \in \{\mathsf{looking}, \mathsf{waiting}\}$. If p is waiting and $P_p = \epsilon$, where $\epsilon \in \mathcal{E}_p$, then we say that p *attends* the committee ϵ. A committee ϵ *meets*, if and only if for every process $p \in \epsilon$, we have $P_p = \epsilon$ and $S_p \in \{\mathsf{waiting}, \mathsf{done}\}$. When a committee ϵ meets, every process $p \in \epsilon$ is *participating in* ϵ. Let $\gamma_0\gamma_1 \dots$ be a computation. We say that a committee meeting ϵ *convenes* in γ_i, where $i > 0$, if and only if ϵ does not meet in γ_{i-1}, but it meets in γ_i. For all $i > 0$, we say that a committee meeting ϵ *terminates* in γ_i, if and only if ϵ meets in γ_{i-1}, but does not meets in γ_i. If a committee meeting ϵ terminates in γ_i, where $i > 0$, then there exists a process p, such that (i) $(P_p = \epsilon \ \wedge \ S_p = \mathsf{done})$ in γ_{i-1}, and (ii) $(P_p = \perp \ \wedge S_p = \mathsf{idle})$ in γ_i. In this case, we say that p *leaves* the committee meeting ϵ on transition $\gamma_{i-1} \mapsto \gamma_i$.

For every process p, we assume the existence of two predicates: $RequestIn(p)$ and $RequestOut(p)$. The predicate $RequestIn(p)$ holds when p (or an application at p) requests the participation of p in a committee meeting. When a committee involving p meets or p is still involved in a meeting that is terminated (in this latter case the predicate $LeaveMeeting(p)$ holds), the predicate $RequestOut(p)$ eventually holds, meaning that p wants to voluntarily stop discussing. Once $RequestOut(p)$ is $true$, it remains $true$ until p becomes idle. Note also that, when necessary, we materialize the assumption on infinite meetings by assuming that, for all processes p:

- If p satisfies $S_p = \mathsf{done}$ but $\neg Meeting(p)$ holds, then the predicate $RequestOut(p)$ eventually holds. Indeed, in this case, the meeting involving p is already terminated.
- However, if p is involved in a meeting, then the meeting never ends. Consequently, $Meeting(p) \Rightarrow \neg RequestOut(p)$ forever.

Remark 2 *Guards of actions $Step_1, Step_{21}, Step_{22}, Step_{31}, Step_{32}$, and $Step_4$ are mutually exclusive at each professor.*

Lemma 1 *Every computation of $\mathcal{CC}1 \circ \mathcal{TC}$ satisfies* Exclusion.

Proof. Let ϵ and ϵ' be two conflicting committees, *i.e.*, $\epsilon \cap \epsilon' \neq \emptyset$. Let p be a process in $\epsilon \cap \epsilon'$. By definition, if ϵ (respectively, ϵ') meets, then $P_p = \epsilon$ (respectively, $P_p = \epsilon'$). Hence, ϵ and ϵ' cannot meet simultaneously. □

Lemma 2 *When committee meeting ϵ convenes, every process $p \in \epsilon$ satisfies $(P_p = \epsilon \ \wedge \ S_p = \mathsf{waiting})$.*

Proof. Consider a committee ϵ that convenes in γ_i. By definition, the committee ϵ meets in γ_i, but not in γ_{i-1}. Moreover, for every $p \in \epsilon$, we have $(P_p = \epsilon \ \wedge \ S_p \in \{\mathsf{waiting}, \mathsf{done}\})$ in γ_i. Also, there must exist a process q in committee ϵ, such that $S_q \in \{\mathsf{idle}, \mathsf{looking}\}$ or $P_q \neq \epsilon$ in γ_{i-1}. We now prove the lemma by contradiction. Assume that there exists process $r \in \epsilon$, such that

$S_r =$ done in γ_i. Then, either (1) $S_r =$ done in γ_{i-1}, or (2) r executes action $Step_{32}$ on transition $\gamma_{i-1} \mapsto \gamma_i$. In case (1), during $\gamma_{i-1} \to \gamma_i$, process q cannot set (S_q,P_q) to:

- (waiting,ϵ), because of the state of r; or
- (done,ϵ), because otherwise $S_q =$ waiting and $P_q = \epsilon$ in γ_{i-1}.

In case (2), ϵ already meets in γ_{i-1} (see Predicate $Meeting(r)$), which is a contradiction. Thus, for every $p \in \epsilon$, we have ($P_p = \epsilon \ \wedge \ S_p =$ waiting) in γ_i and, hence, the lemma holds. □

Corollary 2 *Every computation of $\mathcal{CC}1 \circ \mathcal{TC}$ satisfies Synchronization.*

Lemma 3 *For every process p, if $Correct(p)$ holds, then $Correct(p)$ continues to hold forever.*

Proof. We prove this lemma by showing that if a process p satisfies $Correct(p)$ in some configuration γ, then p satisfies $Correct(p)$ in configuration γ' where $\gamma \mapsto \gamma'$ is a transition.

According to the definition of $Correct$, we distinguish the following four cases in γ:

(a) $S_p =$ *idle* $\wedge$ $P_p = \perp$. Obviously, if p does not modify S_p or P_p in the next step, then $Correct(p)$ holds in the next configuration step as well. Now, the only action modifying S_p and/or P_p that may be enabled in p is $Step_1$. If p executes action $Step_1$, then $P_p :=$ looking and $Correct(p)$ still holds in γ'.

(b) $S_p =$ *looking*. Obviously, if p does not modify S_p in the next step, then $Correct(p)$ holds in the next configuration step as well. Now, suppose that p modifies S_p on transition $\gamma \mapsto \gamma'$. In this case, p has to execute $Step_{31}$. Consequently, in γ we have $P_p = \epsilon$, where $\epsilon \in \mathcal{E}_p$, and, $\forall q \in \epsilon : (P_q = \epsilon \wedge S_q \in \{\text{looking}, \text{waiting}\})$. Now, in this case, every process $q \in \epsilon$ satisfies $Ready(q)$ and $\neg Meeting(q)$. So, no process $q \in \epsilon$ can modify P_q on transition $\gamma \mapsto \gamma'$. Moreover, every process $q \in \epsilon$ can only execute $Step_{31}$ to modify S_q on transition $\gamma \mapsto \gamma'$. Thus, in configuration γ', the predicate $\forall q \in \epsilon : (P_q = \epsilon \wedge S_q \in \{\text{looking}, \text{waiting}\})$ still holds and, as a consequence, $Correct(p)$ holds as well.

(c) $S_p =$ *waiting* $\wedge$ $P_p = \epsilon$*, where* $\epsilon \in \mathcal{E}_p$*.* In this case, $Correct(p)$ implies the following possible subcases in γ:

   (1) $\forall q \in \epsilon : (P_q = \epsilon \wedge S_q \in \{\textit{looking}, \textit{waiting}\}) \ \wedge \ \exists r \in \epsilon : S_r =$ *looking*. In this subcase, every process $q \in \epsilon$ satisfies $Ready(q)$ and $\neg Meeting(q)$. So, no process $q \in \epsilon$ can modify P_q on transition $\gamma \mapsto \gamma'$. Moreover, every process $q \in \epsilon$ can only execute $Step_{31}$ to modify S_q on transition $\gamma \mapsto \gamma'$. Thus, the predicate ($\forall q \in \epsilon : (P_q = \epsilon \wedge S_q \in \{\text{looking}, \text{waiting}\}$) holds in γ' and, as a consequence, $Correct(p)$ holds in γ' as well.

   (2) $\forall q \in \epsilon : (P_q = \epsilon \wedge S_q \in \{\textit{waiting}, \textit{done}\})$. In this subcase, because of the state of p, every process $q \in \epsilon$ satisfies $Meeting(q)$ and $\neg LeaveMeeting(q)$. So, no process $q \in \epsilon$ can modify P_q on transition $\gamma \mapsto \gamma'$. Moreover, every process $q \in \epsilon$ can only execute $Step_{32}$ to modify S_q on transition $\gamma \mapsto \gamma'$. Thus, the predicate ($\forall q \in \epsilon : (P_q = \epsilon \wedge S_q \in \{\text{waiting}, \text{done}\}$ still holds in γ' and, as a consequence, $Correct(p)$ holds as well.

(d) $S_p = \textsf{done} \wedge P_p = \epsilon$, *where* $\epsilon \in \mathcal{E}_p$. In this case, $Correct(p)$ implies the following possible subcases in γ:

(1) $\forall q \in \epsilon : (P_q = \epsilon \wedge S_q \in \{\textsf{waiting}, \textsf{done}\}) \wedge \exists r \in \epsilon : S_r = \textsf{waiting})$. This subcase has been already considered in case $(c).(2)$, so $Correct(p)$ holds in γ'.

(2) $\forall q \in \epsilon : (P_q = \epsilon \Rightarrow S_q = \textsf{done})$. In this case, no process q that satisfies $P_q \neq \epsilon$ can execute $P_q := \epsilon$, because $\epsilon \notin FreeEdges_q$. Also, a process q that satisfies $P_q = \epsilon$ in γ (*e.g.*, p) can only modify P_q and/or S_q by executing action $Step_4$ on transition $\gamma \mapsto \gamma'$. In this case, $S_q := \textsf{idle}$ and $P_q := \perp$. As a consequence, in γ' either $S_p := \textsf{idle}$ and $P_p := \perp$, or $P_p = \epsilon \wedge \forall q \in \epsilon : (P_q = \epsilon \Rightarrow S_q = \textsf{done})$. Thus, $Correct(p)$ holds in γ' as well.

Since in all possible cases, $Correct(p)$ is preserved by the algorithm's actions, the lemma holds. $\square$

It is straightforward to see that a process that satisfies $\neg Correct$ is enabled for either action $Stab_1$ or action $Stab_2$ (the priority actions). Moreover, since the daemon is weakly fair, Lemma 3 implies the following corollary:

Corollary 3 *After at most one round, every process p satisfies $Correct(p)$ forever.*

Lemma 4 *After committee ϵ convenes, the predicate $(\forall p \in \epsilon : (P_p = \epsilon \wedge S_p = \textsf{done}))$ eventually holds.*

Proof. Consider a configuration γ where every process $p \in \epsilon$ satisfies $(P_p = \epsilon \wedge S_p \in \{\textsf{waiting}, \textsf{done}\})$, and, there exists a process $q \in \epsilon$, such that $(P_q = \epsilon \wedge S_q = \textsf{waiting})$. Then, every process $p \in \epsilon$ satisfies $Correct(p)$ in γ and, by Lemma 3, (*) actions $Stab_1$ and $Stab_2$ are disabled forever at every $p \in \epsilon$ from γ. Now, in configuration γ, a process $p \in \epsilon$, where $S_p = \textsf{done}$, cannot modify P_p or S_p. Moreover, in γ, a process $q \in \epsilon$, where $(P_q = \epsilon \wedge S_q = \textsf{waiting})$ cannot modify P_q and can only set S_q to $\textsf{done}$ by executing action $Step_{32}$, which is continuously enabled. Since we assume a weakly fair daemon, q eventually executes action $Step_{32}$ by (*) and Remark 2. Hence, the lemma holds. $\square$

Corollary 4 *Every computation of $\mathcal{CC}1 \circ \mathcal{TC}$ satisfies the* Essential Discussion.

Proof. The proof is trivial by Lemmas 2, 4, and action $Step_{32}$. $\square$

Lemma 5 *Every computation of $\mathcal{CC}1 \circ \mathcal{TC}$ satisfies the* Voluntary Discussion.

Proof. Let a committee ϵ convene in configuration γ_i. By Lemmas 2, every process $p \in \epsilon$ satisfies $Correct(p)$ in γ_i and, by Lemma 3, (*) actions $Stab_1$ and $Stab_2$ are disabled forever at every $p \in \epsilon$. By Corollary 4, every process of committee ϵ eventually executes its essential discussion. Thus, following Lemmas 2 and 4, the system reaches a configuration γ_j $(j > i)$, where every process $p \in \epsilon$ satisfies $(P_p = \epsilon \wedge S_p = \textsf{done})$. In such a configuration, a process p in ϵ can update its P_p and/or S_p only if it satisfies the predicate $RequestOut(p)$. Now, by hypothesis it will happen, and in this case, $Step_4$ will be the priority enabled action at p (by (*)) meaning that it voluntarily decides to leave the meeting. Moreover, by definition, since a process eventually satisfies $RequestOut$ continuously and the daemon is weakly fair, the meeting eventually terminates due to execution of action $Step_4$ by some process. Therefore, the lemma holds. $\square$

Observe that in the algorithm, a process that does not satisfy $Correct$ can only execute either action $Stab_1$ or action $Stab_2$. Thus:

Remark 3 *If a process p is waiting and satisfies $\neg Correct(p)$, it remains waiting (at least) until it satisfies $Correct(p)$.*

Lemma 6 *Every computation of $\mathcal{CC}1 \circ \mathcal{TC}$ satisfies* Progress*.*

Proof. We prove this lemma by contradiction. Suppose there exists a computation c of $\mathcal{CC}1 \circ \mathcal{TC}$ that does not satisfy Progress.

Let $\mathcal{E}_\gamma^\infty$ be the subset of $\mathcal{E}$ such that $\forall \epsilon \in \mathcal{E}$, $\epsilon \in \mathcal{E}_\gamma^\infty$ if and only if for all processes $p \in \epsilon$, p is waiting in γ, but will never more participate in a meeting during c. By definition, $\forall \gamma_i, \gamma_j$ such that γ_j occurs after γ_i in c, we have $\mathcal{E}_{\gamma_i}^\infty \subseteq \mathcal{E}_{\gamma_j}^\infty$. Moreover, the number of processes being finite, there exist configurations γ_i in c such that $\mathcal{E}_{\gamma_i}^\infty = \mathcal{E}_{\gamma_j}^\infty$, for every configuration γ_j that occurs after γ_i in c.

Let now consider such a configuration, say γ^1, and let V^∞ be the subset of all processes that are incident to a hyperedge in $\mathcal{E}_{\gamma^1}^\infty$. We distinguish the following two cases in γ^1:

(a) *There is a process $p \in V^\infty$ that eventually satisfies $Ready(p)$.* This case implies that $P_p = \epsilon$, where $\epsilon \in \mathcal{E}_p$. By definition of $Ready$, every process $q \in \epsilon$ satisfies $(P_q = \epsilon \wedge S_q \in \{\textsf{looking}, \textsf{waiting}\})$, which in turns, implies $Correct(q)$. So, by Lemma 3, (*) actions $Stab_1$ and $Stab_2$ are disabled forever at every $q \in \epsilon$ from γ^1.

Now, observe that in configuration γ^1 a process p in ϵ, where $S_p = \textsf{waiting}$, cannot modify P_p or S_p. Also, every process $q \in \epsilon$ such that $(P_q = \epsilon \wedge S_q = \textsf{looking})$ cannot modify P_q and can only modify S_q by action $Step_{31}$, which is its priority enabled action in γ^1 (by (*) and Remark 2). Hence, as the daemon is weakly fair, the committee meeting ϵ eventually convenes, which is a contradiction.

(b) *No process p of V^∞ eventually satisfies $Ready(p)$.* By Remark 3,

(1) Every p of V^∞ remains waiting forever.

(Indeed, the only way to lose the waiting status is to switch to the meeting status.)

Observe that by definition, we have

(2) $FreeEdges_p \neq \emptyset$.

Again, following Remark 3,

(3) $FreeEdges_p$ is fixed.

By Corollary 3, there exists a configuration γ^2 in c after γ^1 where:

(4) All processes satisfy $Correct$ forever.

By Property 1, eventually there exists a unique token in the network. If a process in V^∞ eventually get the token, then it never releases it by (1), (2), and (3).

Assume now, by the contradiction, that no process in V^∞ eventually gets this token (from γ^2). Assume first that a token holder participates in a meeting. Then it eventually releases the token by Lemma 5. In contrast, if it never more participates in any meeting, then it has status idle forever, so its action $Token_2$ is continuously enabled. As the daemon being weakly fair and $Token_2$ is its priority enabled action (by (4)), the process eventually releases the token. Hence, there exists a configuration γ^3 in c after γ^2 where:

(5) There exists a unique process $\ell \in V^\infty$ that satisfies $Token(\ell)$ forever.

(6) Every process $p \in V \setminus \{\ell\}$ satisfies $\neg Token(p)$ forever.

Every process p having status idle forever and that never gets the token has action $Token_1$ that is continuously enabled (its priority enabled action by (4))) if $T_p = true$. The daemon being weakly fair, eventually satisfies $T_p = false$ forever. Moreover, by definition every other process q in $V \setminus V^\infty$ convenes and terminates meetings infinitely often, and each time q executes $Step_4$, T_q is reset to $false$. Hence, from (5), we can deduce that there exists a configuration γ^4 in c after γ^3 where:

(7) Every process q in $V \setminus V^\infty$ satisfies $\neg T_q$ forever.

By (4) and the fact that no process in V^∞ satisfies $Ready$, we have (in particular, from γ^4):

(8) all processes in V^∞ are in looking status.

Consider then a process q in V^∞ such that $T_q \neq Token(q)$ (from γ^4). Then, q is continuously enabled, by (5) and (6). So, it is eventually selected by the weakly fair daemon. Now, when selected, its actions $Stab_1$ and $Stab_2$ are disabled by (4). Moreover, $Step_{31}$, $Step_{32}$, and $Step_4$ are also disabled at q, otherwise q will lose its looking status, a contradiction to (8). So, q necessarily executes $Token_1$ (*n.b.*, $Token_2$ is disabled at q by (2), (3), and (8)) and there exists a configuration γ^5 in c after γ^4 where:

(9) ℓ satisfies T_ℓ forever.

(10) Every process $q \in V \setminus \{\ell\}$ satisfies $\neg T_q$ forever.

In particular, (8), (9), and (10) hold for all processes incident to a hyperedge of $FreeEdges_\ell$. So, $LocalMax(\ell) = \ell$ and $LocalMax(r) = \ell$, where r is any process incident to a hyperedge of $FreeEdges_\ell$. So, if $P_\ell \notin FreeEdges_\ell$, then action $Step_{21}$ is its priority enabled action (by (4) and Remark 2). ℓ remains enabled until it executes it. So, ℓ eventually does, because the daemon is weakly fair. Hence, eventually $P_\ell = \epsilon$ forever, where $\epsilon \in FreeEdges_\ell$. Then, every process $r \in \epsilon$, such that $P_r = \epsilon$ is disabled forever, because ℓ never satisfies $Ready(\ell)$, by hypothesis. Finally, action $Step_{22}$ is continuously enabled action at every process $s \in \epsilon$

such that $P_s \neq \epsilon$, moreover it is their priority enabled action by (4) and Remark 2. Again, because the daemon is weakly fair, every process s eventually executes it. Hence, eventually ℓ satisfies $Ready(\ell)$, which is a contradiction.

□

Lemma 7 *Every computation of $\mathcal{CC}1 \circ \mathcal{TC}$ satisfies* Maximal Concurrency*.*

Proof. Assume there is a set P_1 of processes that are all in infinite-time meetings. Let P_2 be a set of processes waiting. Let Π be the set of hyperedges whose all incident processes are in P_2. We now prove the lemma by contradiction. Suppose that $\Pi \neq \emptyset$ and no meeting between processes incident to an hyperedge in Π eventually convenes. We distinguish the following two cases:

(a) *There exists a process $p \in P_2$ that eventually satisfies $Ready(p)$.* In this case, using the same reasoning as in case (a) in the proof of Lemma 6, we obtain a contradiction.

(b) *No process in P_2 eventually satisfies $Ready(p)$.* Let p be a process in P_2. In this case, following Remark 3, p must remain waiting forever (the only way to leave the waiting status is to switch to the meeting status). Observe that by definition, $FreeEdges_p \neq \emptyset$. Using the same reasoning as in case (b) of the proof of Lemma 6, there exists a configuration γ in which:

  (1) There exists a process ℓ that satisfies T_ℓ forever.

  (2) Every process $q \in V \setminus \{\ell\}$ satisfies $\neg T_q$ forever.

  (3) Every process in V satisfies $Correct$ forever.

Now, if $\ell \in P_2$, then using the same reasoning as in case (b) of the proof of Lemma 6, we reach a contradiction. If $\ell \notin P_2$, then, let p_{max} be the process of P_2 having the greatest identifier. Then using the reasoning similar to the case (b) in the proof of Lemma 6 (p_{max} has the same role as ℓ in the proof of Lemma 6), we reach a contradiction.

□

Theorem 2 *The composition $\mathcal{CC}1 \circ \mathcal{TC}$ is a snap-stabilizing algorithm that solves the 2-phase committee coordination problem and satisfies* Maximal Concurrency*.*

Proof. Given Lemmas 1-7, the proof of the theorem trivially follows. □

5. Snap-Stabilizing 2-Phase Committee Coordination with Fairness

We now consider the 2-phase committee coordination problem in systems where processes are waiting for meetings infinitely often. In such a setting, an idle process always eventually becomes waiting. Hence, for simplicity (and without loss of generality), we assume that processes are always requesting when they are not in a meeting. As a consequence, the predicate $RequestIn(p)$ and the state idle are implicit in the actions of the next algorithm. In Subsection 5.1, we present a snap-stabilizing algorithm that guarantees the properties of 2-phase committee coordination and Professor Fairness. The proof of correctness of the algorithm is presented in Subsection 5.2. Then, in Subsection 5.3, we analyze the complexity of our algorithm. Finally, we discuss Committee Fairness in Subsection 5.4.

5.1. Algorithm

Our algorithm is the composite algorithm $\mathcal{CC}2 \circ \mathcal{TC}$, where (1) $\mathcal{CC}2$ is a Snap-stabilizing algorithm that ensures Exclusion, Synchronization, and 2-Phase Discussion, and (2) $\mathcal{TC}$ is the same self-stabilizing module that manages a circulating token as in Section 4. It ensures Fairness, and consequently Progress.

Algorithm $\mathcal{CC}2$ is identical for all processes in the distributed system. Its code is given in Algorithm 2. Similar to Algorithm $\mathcal{CC}1$, each process p maintains S_p, P_p, and T_p with the same meaning. Also, the token defines priorities to convene committees. However, to guarantee fairness, in this algorithm, a token is released only when its holder leaves a meeting.

After receiving a token, a looking process p selects a smallest (in terms of members) incident committee ϵ (this constraint is used only to slightly enhance the concurrency) using its edge pointer P_p ($Step_{11}$). Note that unlike the previous algorithm, the members of the chosen committee are not necessarily all looking. Then, process p sticks with committee ϵ until ϵ convenes. By assumption, other members of committee ϵ are eventually looking and, hence, ϵ is selected by action $Step_{12}$.

In order to obtain the best concurrency as possible (recall that maximal concurrency is impossible in this case), a process that is not in a committee ϵ must not wait for a process involved in ϵ. To that goal, we introduce the Boolean variable L, which shows whether or not a process is *locked*. A locked process is one that is incident to a hyperedge that contains a process that (1) owns the token, (2) has set its pointer to that hyperedge, and (3) is looking to start a committee meeting. The locks are maintained using action $Lock$. Hence, processes that are not in ϵ try to convene committees that do not involve *locked* processes ($Step_{13}$ and $Step_{14}$). As in Algorithm $\mathcal{CC}1$, we use the process identifiers to define priorities among the looking processes not in ϵ. The rest of actions of the algorithm are similar to those of Algorithm $\mathcal{CC}1$.

Figure 4 illustrates the need of the Boolean L. In this configuration, Professor 8 chooses the committee $\{1, 2, 5, 8\}$ because Professor 1 has the token. Moreover, this committee cannot meet before the meeting of committee $\{3, 4, 5\}$ terminates. Now, to ensure fairness, Professors 1, 2, and 8 should not change their P-pointers so that eventually a meeting of $\{1, 2, 5, 8\}$ convenes. Furthermore, to obtain a better concurrency, Committee $\{6, 7, 9\}$ should be allowed to meet. Now, for Professor 9, Committee $\{8, 9\}$ has higher priority than Committee $\{6, 7, 9\}$. By definition, all members of Committee $\{1, 2, 5, 8\}$ are locked. So, thank to the Boolean L_8, Professor 9 realizes

Algorithm 2 Pseudo-code of $\mathcal{CC}2$ for process p.

Inputs:

$RequestOut(p)$	:	Predicate: input from the system indicating desire for leaving a committee
$Token(p)$	:	Predicate: input from $\mathcal{TC}$ indicating process p owns the token
$ReleaseToken_p$	:	Statement: output to $\mathcal{TC}$ indicating process p releases the token

Constant:

$\mathcal{E}_p$: Set of hyperedges incident to p

Variables:

T_p, L_p	:	Booleans
$P_p \in \mathcal{E}_p \cup \{\bot\}$	:	Edge pointer
$S_p \in \{\mathsf{looking}, \mathsf{waiting}, \mathsf{done}\}$	:	Status

Macros:

$$FreeEdges_p = \{\epsilon \in \mathcal{E}_p \mid \forall q \in \epsilon \ : \ (S_q = \mathsf{looking} \ \wedge \ \neg L_q \ \wedge \ \neg T_q)\}$$
$$FreeNodes_p = \{q \mid \exists \epsilon \in FreeEdges_p \ : \ q \in \epsilon\}$$
$$TPointingEdges_p = \{\epsilon \in \mathcal{E}_p \mid \exists q \in \epsilon \ : \ (P_q = \epsilon \ \wedge \ T_q \wedge \ S_q = \mathsf{looking})\}$$
$$TPointingNodes_p = \{q \mid \exists \epsilon \in TPointingEdges_p \ : \ q \in \epsilon\}$$
$$MinSize_p = \min_{\epsilon \in \mathcal{E}_p} |\epsilon|$$
$$MinEdges_p = \{\epsilon \in \mathcal{E}_p \ \mid \ |\epsilon| = MinSize_p\}$$

Predicates:

$$Locked(p) \equiv TPointingEdges_p \neq \emptyset$$
$$Ready(p) \equiv \exists \epsilon \in \mathcal{E}_p \ : \ \forall q \in \epsilon \ : \ (P_q = \epsilon \ \wedge \ S_q \in \{\mathsf{looking}, \mathsf{waiting}\})$$
$$Meeting(p) \equiv \exists \epsilon \in \mathcal{E}_p \ : \ \forall q \in \epsilon \ : \ (P_q = \epsilon \ \wedge \ S_q \in \{\mathsf{waiting}, \mathsf{done}\})$$
$$LeaveMeeting(p) \equiv \exists \epsilon \in \mathcal{E}_p \ : \ (P_p = \epsilon \ \wedge \ S_p = \mathsf{done} \ \wedge \ (\forall q \in \epsilon \ : \ (P_q = \epsilon \ \Rightarrow \ S_q \neq \mathsf{waiting})))$$
$$LocalMax(p) \equiv p = \max(FreeNodes_p)$$
$$MaxToFreeEdge(p) \equiv \neg Token(p) \ \wedge \ \neg Locked(p) \ \wedge \ FreeEdges_p \neq \emptyset \ \wedge \ LocalMax(p) \ \wedge \ \neg Ready(p) \ \wedge \ P_p \notin FreeEdges_p$$
$$JoinLocalMax(p) \equiv \neg Token(p) \ \wedge \ \neg Locked(p) \ \wedge \ FreeEdges_p \neq \emptyset \ \wedge \ \neg LocalMax(p) \ \wedge \ \neg Ready(p) \ \wedge \ \exists \epsilon \in FreeEdges_p \ : \ (P_{\max(FreeNodes_p)} = \epsilon \wedge P_p \neq \epsilon)$$
$$TokenHolderToEdge(p) \equiv Token(p) \ \wedge \ (S_p = \mathsf{looking}) \ \wedge \ \neg Ready(p) \ \wedge \ (P_p \notin MinEdges_p)$$
$$JoinTokenHolder(p) \equiv \neg Token(p) \ \wedge \ (S_p = \mathsf{looking}) \ \wedge \ \neg Ready(p) \ \wedge \ Locked(p) \ \wedge \ (P_p \notin TPointingEdges_p)$$
$$Correct(p) \equiv [(S_p = \mathsf{waiting}) \ \Rightarrow \ Ready(p) \ \vee \ Meeting(p)] \ \wedge \ [(S_p = \mathsf{done}) \ \Rightarrow \ Meeting(p) \ \vee LeaveMeeting(p)]$$

Actions:

$Lock$	::	$Locked(p) \neq L_p$	$\mapsto$	$L_p := Locked(p)$;
$Step_{11}$	::	$TokenHolderToEdge(p)$	$\mapsto$	$P_p := \epsilon$ such that $\epsilon \in MinEdges_p$;
$Step_{12}$	::	$JoinTokenHolder(p)$	$\mapsto$	$P_p := \epsilon$ such that $\epsilon \in \mathcal{E}_p$, where $P_{\max(TPointingNodes_p)} = \epsilon$;
$Step_{13}$	::	$MaxToFreeEdge(p)$	$\mapsto$	$P_p := \epsilon$ such that $\epsilon \in FreeEdges_p$;
$Step_{14}$	::	$JoinLocalMax(p)$	$\mapsto$	$P_p := \epsilon$ such that $\epsilon \in \mathcal{E}_p$, where $P_{\max(FreeNodes_p)} = \epsilon$;
$Token$	::	$Token(p) \neq T_p$	$\mapsto$	$T_p := Token(p)$;
$Step_2$	::	$Ready(p) \wedge (S_p = \mathsf{looking})$	$\mapsto$	$S_p := \mathsf{waiting}$;
$Step_3$	::	$Meeting(p) \wedge (S_p = \mathsf{waiting})$	$\mapsto$	$\langle \mathsf{EssentialDiscussion} \rangle; S_p := \mathsf{done}$;
$Step_4$	::	$LeaveMeeting(p) \wedge RequestOut(p)$	$\mapsto$	$S_p := \mathsf{looking}; P_p := \bot; T_p := false$; **if** $Token(p)$ **then** $ReleaseToken_p$ **fi**;
$Stab$	::	$\neg Correct(p)$	$\mapsto$	$S_p := \mathsf{looking}; P_p := \bot$;

that he should not give priority to $\{8, 9\}$. Consequently, he will select $\{6, 7, 9\}$ by action $Step_{13}$, improving concurrency.

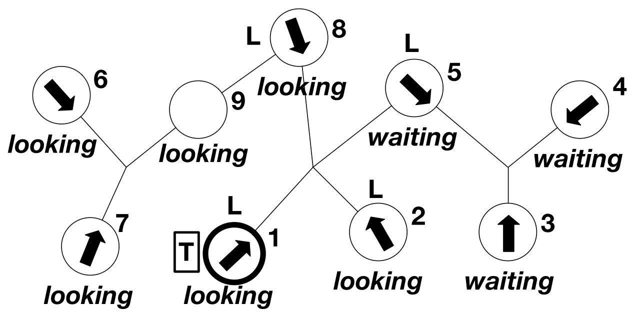


Figure 4: Example of locked professors.

5.2. Correctness of $\mathcal{CC}2 \circ \mathcal{TC}$

We recall that in the following proofs, we assume that computations of $\mathcal{CC}2 \circ \mathcal{TC}$ start from an arbitrary configuration. In the proofs below we use some notions and terminology already defined in Subsection 4.2.

Following a similar approach to the one used in Subsection 4.2, we have the following technical results:

Remark 4 *Guards of actions $Step_{11}$, $Step_{12}$, $Step_{13}$, $Step_{14}$, $Step_2$, $Step_3$, and $Step_4$ are mutually exclusive at each professor.*

Lemma 8 *For every process p, if $Correct(p)$ holds, then $Correct(p)$ holds forever.*

Corollary 5 *After at most one round, every process p satisfies $Correct(p)$ forever.*

From these technical results, we can deduce the following lemma using the same reasoning as in Subsection 4.2.

Lemma 9 *Every computation of $\mathcal{CC}2 \circ \mathcal{TC}$ satisfies:*

1. *Exclusion,*
2. *Synchronization,*
3. *Essential Discussion, and*
4. *Voluntary Discussion.*

We now focus on the Professor Fairness.

Lemma 10 *From any configuration where every process q satisfies $Correct(q)$, we have: if a process p that satisfies $Ready(p)$, $Meeting(p)$, or $S_p =$ done, then p eventually executes action $Step_4$.*

Proof. Observe that from such a configuration, (*) *every process* q *satisfies* $Correct(q)$ *forever* by Lemma 8. As a consequence, from that point every process p that satisfies $Ready(p)$, $Meeting(p)$, or $S_p = \mathsf{done}$ satisfies one of the following cases:

- $LeaveMeeting(p)$ ***holds.*** In this case, $S_p = \mathsf{done}$ and $P_p \neq \perp$. Let ϵ be the value of P_p. $S_p = \mathsf{done}$ implies $\neg Ready(p)$. So, while $S_p = \mathsf{done}$, no process q can execute $Step_2$ to then satisfy $P_q = \epsilon \wedge S_q = \mathsf{waiting}$. Also, every process q that satisfies $P_q = \epsilon \wedge S_q = \mathsf{done}$ can only update S_q and/or P_q by executing action $Step_4$ by (*), that is $S_q := \mathsf{looking}$ and $P_q := \perp$. As a consequence, while p does not execute action $Step_4$, $LeaveMeeting(p)$ holds. Now $RequestOut(p)$ eventually continuously holds, and, thus, action $Step_4$ is eventually continuously enabled at p. As the daemon is weakly fair, p is eventually selected to execute an action, and this action is $Step_4$ by (*), which proves the lemma in this case.

- $Meeting(p) \wedge \neg LeaveMeeting(p)$ ***holds.*** Then, $Meeting(p)$ implies that $P_p \neq \perp$. Let ϵ be the value of P_p. No process $r \in \epsilon$ can update P_r. Moreover, for every process $r \in \epsilon$, r can modify its status S_r only if $S_r = \mathsf{waiting}$. Now, $Step_3$ is enabled at every of those processes, and this action is their priority enabled action by (*) and Remark 4. Observe that $(Meeting(p) \wedge \neg LeaveMeeting(p))$ holds until all these processes have moved and, as the daemon is weakly fair, they eventually move. At this point this case can be reduced to the previous case, which proves the lemma in this case.

- $Ready(p) \wedge \neg Meeting(p)$ ***holds.*** Then, $Ready(p)$ implies that $P_p \neq \perp$. Let ϵ be the value of P_p. No process $r \in \epsilon$ can update P_r. Moreover, for every process $r \in \epsilon$, r can modify its status S_r only if $S_r = \mathsf{looking}$. Now, $Step_2$ is enabled at every of those processes, and this action is their priority enabled action by (*) and Remark 4. Observe that $Ready(p) \wedge \neg Meeting(p)$ holds until all these processes have moved and, as the daemon is weakly fair, they eventually move. At this point this case can be reduced to the previous case, which proves the lemma in this case.

Thus, in any case, p eventually executes $Step_4$ and the lemma holds. □

Lemma 11 *In every computation of* $\mathcal{CC}2 \circ \mathcal{TC}$*, no process can hold a token forever.*

Proof. By Property 1, the system eventually reaches a configuration from which there is a unique token forever. Assume, by the contradiction, that after such a configuration, some process ℓ holds the unique token forever, *i.e.* $Token(\ell)$ holds forever and for every process $p \neq \ell$, $\neg Token(p)$ holds forever.

Then, using the same reasoning as in case (b) of the proof of Lemma 6, we can deduce that the system reaches a configuration γ from which:

(1) ℓ satisfies $Token(\ell) \wedge T_\ell$ forever.

(2) Every process $p \neq \ell$ satisfies $\neg Token(p) \wedge \neg T_p$ forever.

(3) Every process satisfies $Correct$ forever.

Let us study the following two cases:

(a) *From γ, $S_\ell =$ done, $Ready(\ell)$, or $Meeting(\ell)$ eventually holds.* In this case, we obtain a contradiction by Lemma 10.

(b) *From γ, $S_\ell \neq$ done, $\neg Ready(\ell)$, and $\neg Meeting(\ell)$ hold forever.* We study the following two subcases:

- $P_\ell \in MinEdges_\ell$. In this subcase, by (3), we deduce that $S_\ell =$ looking and $P_\ell \in MinEdges_\ell$ hold forever.

  Then, let ϵ be the hyperedge pointed by P_ℓ. By (1) and (2), we have ($TPointingEdges_p$, $Locked(p)$) that is equal to ($\{\epsilon\}$, $true$) forever for every process $p \in \epsilon$ such that $p \neq \ell$.

  If p satisfies ($S_p =$ looking $\wedge$ $\neg Ready(p)$), eventually $P_p = \epsilon$ because of the weakly fair daemon and action $Step_{12}$ (by (3) and Remark 4, p executes $Step_{12}$ when selected by the daemon). Then, p becomes disable forever because $\neg Ready(\ell)$ holds forever.

  If p satisfies ($S_p \neq$ looking $\vee$ $Ready(p)$), then ($S_p =$ done $\vee$ $Ready(p)$ $\vee$ $Meeting(p)$) holds by (3). By Lemma 10, p eventually satisfies ($S_p =$ looking $\wedge$ $\neg Ready(p)$), and we retrieve the previous case. So eventually $P_p = \epsilon$ and p becomes disabled forever.

  Hence, we can conclude that eventually $P_p = \epsilon$ holds for every process $p \in \epsilon$, that is $Ready(\ell)$, which is a contradiction.

- $P_\ell \notin MinEdges_\ell$. In this subcase, by (3) and the fact that $S_\ell =$ done $\vee$ $Ready(\ell)$ $\vee$ $Meeting(\ell)$ never holds, we can deduce that $S_\ell =$ looking holds forever. Hence, by (1), action $Step_{11}$ is continuously enabled at ℓ, as the daemon is weakly fair, ℓ eventually executes an enabled action. This action is $Step_{11}$ by (3) and Remark 4, and we retrieve the previous case, which leads to a contradiction.

□

We now deduce the next corollary from Property 1 and Lemma 11:

Corollary 6 *In every computation of $\mathcal{CC}2 \circ \mathcal{TC}$, every process holds a token infinitely many times.*

Lemma 12 *Every computation of $\mathcal{CC}2 \circ \mathcal{TC}$ satisfies* Professor Fairness*.*

Proof. Assume by contradiction that eventually some process p stops participating in any meeting. In this case, it no more executes action $Step_3$. This means, in particular, that the process no more executes $S_p :=$ done. As a consequence, it eventually no more executes action $Step_4$. In particular, it eventually no more executes $ReleaseToken_p$, which contradicts Property 1 and Corollary 6. □

By Lemma 9, 12, and the fact that fairness implies progress, we have:

Theorem 3 *The composition $\mathcal{CC}2 \circ \mathcal{TC}$ is a snap-stabilizing algorithm that solves the 2-phase committee coordination problem and satisfies* Professor Fairness*.*

5.3. *Complexity Analysis*

We now analyze the degree of fair concurrency of Algorithm $\mathcal{CC}2 \circ \mathcal{TC}$. To this end, we recall some concepts from graph theory. A *matching* in a hypergraph $\mathcal{H} = (V, \mathcal{E})$ is a subset S of hyperedges of $\mathcal{H}$, such that no two hyperedges in S have a vertex in common. We denote by $\mathcal{M}_{\mathcal{H}}$ the set of all possible matchings of a hypergraph $\mathcal{H}$. The size of a matching is the number of hyperedges that it contains. A *maximal matching* of $\mathcal{H}$ is a matching of $\mathcal{H}$ that has no superset which is a matching of $\mathcal{H}$. We denote by $\mathcal{MM}_{\mathcal{H}}$ the set of all maximal matchings of a hypergraph $\mathcal{H}$. As $\mathcal{H}$ is clear from the context, we omit it from $\mathcal{M}$ and $\mathcal{MM}$. Obviously, $\mathcal{MM} \subseteq \mathcal{M}$.

Observe that by definition, the degree of fair concurrency d satisfies $1 \leq d \leq \min_{\mathcal{MM}}$, where $\min_{\mathcal{MM}}$ is the size of the smallest maximal matching. The *length* of a hyperedge ϵ (denoted by $|\epsilon|$) is the number of nodes incident to ϵ. For every process p, we denote by $\mathcal{E}_p^{\min}$ the subset of hyperedges incident to p of minimum length, *i.e.*, $\epsilon \in \mathcal{E}_p^{\min}$ if and only if $\epsilon \in \mathcal{E}_p$ and $\forall \epsilon' \in \mathcal{E}_p$, $|\epsilon| \leq |\epsilon'|$. Let $\min_{\mathcal{E}_p}$ denote the minimum length of a hyperedge incident to p. Let $MaxMin = \max_{p \in V}(\mathcal{E}_p^{\min})$.

We denote by $\mathcal{H}_{\overline{Y}}$ the subhypergraph induced by $V \setminus Y$. Given a hyperedge ϵ and a vertex p, we define $Y_{\epsilon,p} = \{y \in 2^{\epsilon} \mid p \in y \ \wedge \ |y| < |\epsilon|\}$. Let $Almost(\epsilon, X)$, where ϵ is a hyperedge and X is a set of vertices, be the set $\{m \in \mathcal{MM}_{\mathcal{H}_{\overline{X}}} \mid \forall q \in \epsilon \setminus X \ : \ q \text{ is incident to a hyperedge of } m\}$. Let $\mathcal{AMM}(p) = \bigcup_{\epsilon \in \mathcal{E}_p^{\min}} \bigcup_{y \in Y_{\epsilon,p}} Almost(\epsilon, y)$, where p is a vertex. Let $\mathcal{AMM} = \bigcup_{p \in V} \mathcal{AMM}(p)$. Observe that $\mathcal{AMM}$ may be equal to the emptyset, *e.g.*, when there is only one hyperedge in $\mathcal{H}$.

The set $\mathcal{AMM}$ as defined above characterizes the cases where Professor Fairness and Maximal Concurrency exhibit their conflicting natures. Consider the case where a process p is the token holder and cannot participate in a meeting. In this case, there exists a neighbor of p, say q, in a smallest hyperedge ϵ incident to p, such that q is participating in another committee meeting. It follows that processes in ϵ (including p) that are currently not meeting are blocked until ϵ convenes. This implies that the current setting does not form a maximal matching and, hence, maximal concurrency cannot be achieved. Thus, in order to analyze the Degree of Fair Concurrency, one needs to consider the set of all maximal matchings of the subhypergraph induced by removing those blocked processes.

We formally characterize the degree of fair concurrency of our algorithm in Theorem 4. We obtain this theorem thanks to several technical results proven below.

Lemma 13 *If committee meetings never terminate, the system eventually reaches a configuration from which some process p is the unique token holder forever.*

Proof. First, the system eventually reaches a configuration from which there is a unique token forever, by Property 1. Assume, by contradiction, that this token moves infinitely many times. Then, infinitely many actions $Step_4$ are executed. The number of processes being finite, there is a process q that executes infinitely many actions $Step_4$. After executing $Step_4$, $S_q =$ looking. Now, before executing $Step_4$ again, q must execute $Step_2$ followed by $Step_3$ to go through status done. Now, in that case, a meeting of a committee whose q is member convenes and that meeting never terminates, by hypothesis. So, q cannot execute $Step_4$ ever in that case, because otherwise it would cause the termination of a meeting, and we obtain a contradiction. □

Lemma 14 *If committee meetings never terminate, the system eventually reaches a configuration γ from which for every process p, $S_p =$ **done** $\Rightarrow Meeting(p)$.*

Proof. Let $c = \gamma_0, ...$ be a computation. The number of processes being finite, assume, by contradiction, that there is a process p such that p satisfies $S_p =$ done $\wedge \neg Meeting(p)$ in infinitely many configurations of c, while committee meetings never terminate. Consider the following two cases:

- There exists i such that $\forall j \geq i$, $S_p =$ done $\wedge \neg Meeting(p)$ in γ_j. Then, by Corollary 5, p eventually satisfies $Correct(p)$ forever, which implies that p eventually satisfies $LeaveMeeting(p)$ forever. Moreover, p eventually satisfies $RequestOut(p)$ continuously. Hence, as the daemon is weakly fair, p eventually executes $Step_4$, and we obtain a contradiction.

- *There exists infinitely many steps $\gamma_i \mapsto \gamma_{i+1}$ of c where $S_p =$ done $\wedge \neg Meeting(p)$ in γ_i and $S_p \neq$ done $\vee$ $Meeting(p)$ in γ_{i+1}.* In this case, p participates infinitely many times in meetings that convene and then terminate, a contradiction.

□

Following a similar reasoning, we have:

Lemma 15 *If committee meetings never terminate, the system eventually reaches a configuration γ from which for every process p, $S_p \neq$ waiting.*

From Lemmas 14 and 15, we have the following corollary:

Corollary 7 *If committee meetings never terminate, the system eventually reaches a configuration γ from which for every process p, either $S_p =$ looking forever, or $S_p =$ done forever.*

Lemma 16 *If committee meetings never terminate, then the system eventually reaches a configuration γ from which there is some process ℓ such that:*

1. *ℓ is the only token holder forever.*
2. *$T_\ell = true$ forever.*
3. *Every process $p \neq \ell$ satisfies $T_p = false$ forever.*
4. *There exists $\epsilon \in \mathcal{E}_\ell$ such that:*
   (a) *$P_\ell = \epsilon$ forever.*
   (b) *$\forall p \in \epsilon$, $L_p = true$ forever.*
   (c) *$\forall p \in V \setminus \epsilon$, $L_p = false$ forever.*

Proof. Case 1 follows from Lemma 13.

Consider Cases 2 and 3. From case 1, we know that for every process p, the value of $Token(p)$ does not change anymore. So, if p satisfies $T_p \neq Token(p)$, then this remains true until p executes action $Token$. Now, eventually actions $Stab$, $Step_2$, $Step_3$, and $Step_4$ are disabled forever at p by Corollaries 5, 7, and Remark 4. So, eventually, p is selected by the daemon to execute action $Token$. Hence, eventually, the value of T_p is fixed and $T_p = Token(p)$ forever.

Consider now case 4a. Eventually the system reaches a configuration from which (*) every process p satisfies $Correct(p)$ forever (by Corollary 5), $S_p = \mathsf{done} \Rightarrow Meeting(p)$ (by Lemma 14), and either $S_p = \mathsf{looking}$ forever, or $S_p = \mathsf{done}$ forever (by Corollary 7).

From such a configuration:

- If $S_\ell = \mathsf{done}$, then ℓ is in an infinite meeting and consequently, there exists $\epsilon \in \mathcal{E}_\ell$ such that $P_\ell = \epsilon$ forever.

- Otherwise, $S_\ell = \mathsf{looking}$ and $Token(\ell)$ holds forever by 1. If ℓ eventually satisfies $Ready(\ell)$, p can execute $Step_2$ by (*) and Remark 4, a contradiction to Corollary 7. So, $\neg Ready(\ell)$ forever and we have either $P_\ell \in MinEdges_p$ and P_ℓ is fixed to that value forever; or, action $Step_{11}$ is continuously enabled. In this latter case, the daemon being weakly fair, ℓ eventually executes $Step_{11}$ (by (*), 2, and Remark 4) and we retrieve the previous case.

Hence case 4a holds in both cases.

Finally, consider Cases 4b and 4c. Let p be process. From γ, if eventually $L_p = Locked(p)$ holds, then L_p is fixed forever by 2, 4a, and Corollary 7. In this case, p satisfies Cases 4b and 4c.

Otherwise, eventually actions $Stab$, $Step_2$, $Step_3$, and $Step_4$ are eventually disabled forever at p by Corollary 5 and Corollary 7. By 2 and 3, action $Token$ is also eventually disabled forever. From that point, p can execute actions $Step_{11}$ to $Step_{14}$ at most once before some neighboring process executes action $Lock$ to definitely fix the value of its variable L. So, as the number of neighbors is finite, action $Lock$ is eventually the only action that p can execute. Thus, as the daemon is weakly fair, p eventually execute action $Lock$ and we retrieve the previous case. □

Lemma 17 *If committee meetings never terminate, the system eventually reaches a configuration γ where $FreeEdges_p = \emptyset$ forever for all processes p.*

Proof. Consider a computation $c = \gamma_0 \dots$ where committee meetings never terminate.

Then, the system eventually reaches configuration from which: for every process p, the value of $FreeEdges_p$ is fixed and $Correct(p) = true$ forever by Lemma 16, Corollaries 5, and 7.

Assume that, from such a configuration, $FreeEdges \neq \emptyset$ for some processes. Let q be the one among those processes with the highest identity. $\forall \epsilon \in FreeEdges_q$, $\forall s \in \epsilon$, $LocalMax(s) = q$ (in particular $LocalMax(q) = q$) holds continuously until a meeting involving q convenes, by Lemma 16. Then, by definition of action $Step_{13}$, Remark 4, and the fact that the daemon is weakly fair, q eventually sticks its pointer on some hyperedge ϵ of $FreeEdges_q$ and then eventually satisfies $Ready(q)$ by definition of action $Step_{14}$. Then, again by definition of action $Step_2$, Remark 4, and the fact that the daemon is weakly fair, some process of ϵ eventually executes action $Step_2$, a contradiction to Corollary 7.

Hence, eventually every process r satisfies $FreeEdges_r = \emptyset$ forever. □

Theorem 4 *Degree of Fair Concurrency of Algorithm $\mathcal{CC}2 \circ \mathcal{TC}$ is at least* $\min_{\mathcal{MM} \cup \mathcal{AMM}}$.

Proof. If committee meetings never terminate, the system eventually reaches a configuration γ where:

1. Every process s satisfies:
   (a) $\mathit{FreeEdges}_s = \emptyset$ (Lemma 17).
   (b) $S_s = \mathsf{looking}$ if and only if s is not in any meeting (Corollary 5 and Lemma 14).
2. By Lemma 16, there is a unique process ℓ such that:
   (a) ℓ is the only token holder forever.
   (b) $T_\ell = \mathit{true}$ forever.
   (c) Every process $p \neq \ell$ satisfies $T_\ell = \mathit{false}$ forever.
   (d) There exists $\epsilon \in \mathcal{E}_\ell$ such that:
      i. $P_\ell = \epsilon$ forever.
      ii. $\forall p \in \epsilon$, $L_p = \mathit{true}$ forever.
      iii. $\forall p \in V \setminus \epsilon$, $L_p = \mathit{false}$ forever.

Consider the following two cases in γ:

- *ℓ participates in a meeting ϵ.* Let r be a process that does not participate in a meeting in γ. Then, eventually $\mathit{FreeEdges}_r = \emptyset$ by case 1a. In this case, for each hyperedge ϵ' incident to r, there a process $t \in \epsilon'$, such that T_t, L_t, or $S_t \neq \mathsf{looking}$ holds. In the two first cases, t participates in the meeting ϵ by case 2. In the latter case, t participates in another meeting by case 1b.

  It follows that for all processes r that is not in a meeting in γ and for all hyperedges ϵ' incident to r, there exists a process in ϵ' that participates in a meeting in γ. Hence, the meetings that hold in γ form a maximal matching of the underlying hypergraph $\mathcal{H}$.

- *ℓ does not participate in any meeting.* In γ, $P_\ell = \epsilon$ such that $\epsilon \in \mathcal{E}_\ell^{\min}$ (see action Step_{13}). Also, there is at least one neighbor of ℓ that participates in a meeting in γ. Let X be the subset of processes in ϵ that do not participate in a meeting in γ. Then, $X \subset \epsilon$ and $\ell \in X$. Following a reasoning similar to the previous case, we can deduce that for all processes s that is not in a meeting in γ and for all hyperedges ϵ' incident to s, there exists a process in ϵ' that either participates in a meeting in γ or is a process of X. Hence, the meetings that hold in γ form a maximal matching of $\mathit{Almost}(\epsilon, X)$.

Hence, the meetings that hold in γ form a matching of $\mathcal{MM} \cup \mathcal{AMM}$. □

In the next theorem, we present a lower bound for $\min_{\mathcal{MM} \cup \mathcal{AMM}}$.

Theorem 5 $\min_{\mathcal{MM} \cup \mathcal{AMM}} \geq (\min_{\mathcal{MM}} - \mathit{MaxMin} + 1)$.

Proof.

- By definition $\mathit{MaxMin} > 0$. So, $\min_{\mathcal{MM}} \geq \min_{\mathcal{MM}} - \mathit{MaxMin} + 1$.

- Let x be the size of the smallest matching in $\mathcal{AMM}$. By definition, there exists a process p, a hyperedge $\epsilon \in \mathcal{E}_p^{\min}$, and a set of processes X where $X \subset \epsilon$ and $p \in X$, such that there exists a maximal matching S of $\mathit{Almost}(\epsilon, X)$ of size x. By definition, S is a matching of $\mathcal{H}$. Moreover, there exists a maximal matching S' of $\mathcal{H}$ such that $S \subset S'$. By definition there exists at most one hyperedge of S' incident to some process in X. Hence, $|S| \geq |S'| - |X|$, *i.e.*, $|S| \geq |S'| - |\epsilon| + 1$, which in turn implies that $|S| \geq \min_{\mathcal{MM}} - |\epsilon| + 1$. It follows that $|S| \geq \min_{\mathcal{MM}} - MaxMin + 1$. Hence, the size of the smallest matching in $\mathcal{AMM}$ is at least $\min_{\mathcal{MM}} - MaxMin + 1$.

□

To evaluate Waiting Time of $\mathcal{CC}2 \circ \mathcal{TC}$, we need to introduce $\max_{Disc}$ which is the maximum amount of rounds a process discusses in a meeting. We assume that $\mathcal{TC}$ is a fair composition of the token circulation algorithm in [27] and the leader election algorithm in [23]. It follows that the following properties hold: (1) starting from any configuration, there is a unique token in the distributed system in $O(n)$ rounds, and (2) once there is a unique token, $O(n)$ processes can receive the token before a process receives the token.

Theorem 6 *In Algorithm $\mathcal{CC}2 \circ \mathcal{TC}$, the worst case Waiting Time is $O(\max_{Disc} \times n)$ rounds, where n is the number of processes.*

Proof. First, from [27, 23], Corollary 5, and Property 1, we know that starting from any arbitrary configuration, the system reaches a configuration γ from where every process satisfies $Correct$ and there is one token forever in $O(n)$ rounds. Now, consider a token holder p in any configuration that follows γ, where p satisfies one of the following three cases:

- $S_p =$ *done*. In this case, in at most one round, p satisfies $LeaveMeeting(p)$ and at most $\max_{Disc}$ rounds later, it is enabled to execute $Step_4$. Hence, p releases the token in $O(\max_{Disc})$ rounds.

- $S_p =$ *waiting*. In this case, in at most one round, p satisfies $Meeting(p)$ and after one more round, it satisfies $S_p =$ done. Hence, from the previous case, we can deduce that p releases the token in $O(\max_{Disc})$ rounds.

- $S_p =$ *looking*. In this case, in one round p sets T_p to true. One another round later, p sets P_p to ϵ where $\epsilon \in \mathcal{E}_p^{\min}$. After this round and similarly to the previous case, every other process in ϵ that was in a meeting, leaves its meeting and joins meeting ϵ in $O(\max_{Disc})$ rounds, which leads to the status $S_p =$ waiting in the next round. Hence, from the previous cases, we can deduce that p releases the token in $O(\max_{Disc})$ rounds.

It follows that after $O(n)$ rounds, a process can keep the token for $O(\max_{Disc})$ consecutive rounds before releases it. Now, from [27, 23], we know that $O(n)$ processes can hold the token before a given process receives it. Hence, the Waiting Time is $O(\max_{Disc} \times n)$ rounds.

□

5.4. Committee Fairness

Algorithm $\mathcal{CC}2 \circ \mathcal{TC}$ can be easily modified to satisfy the Committee Fairness as follows. Every time a process acquires the token, it sequentially selects a new incident committee. This way, we obtain an algorithm, called Algorithm $\mathcal{CC}3 \circ \mathcal{TC}$ that satisfies Committee Fairness. Waiting Time of this algorithm remains the same as that of Theorem 6, but Degree of Fair Concurrency will be slightly degraded. Recall that $Y_{\epsilon,p} = \{y \in 2^{\epsilon} \mid p \in y \ \wedge \ |y| < |\epsilon|\}$. Now, we let $\mathcal{AMM}'(p) = \bigcup_{\epsilon \in \mathcal{E}_p} \bigcup_{y \in Y_{\epsilon,p}} Almost(\epsilon, y)$ and $\mathcal{AMM}' = \bigcup_{p \in V} \mathcal{AMM}'(p)$. Also, let $MaxHEdge = \max_{\epsilon \in \mathcal{E}} |\epsilon|$.

Following a proof similar to the one of Theorem 4, we trivially obtain the proof of the following theorem.

Theorem 7 *The degree of fair concurrency of Algorithm $\mathcal{CC}3 \circ \mathcal{TC}$ is at least* $\min_{\mathcal{MM} \cup \mathcal{AMM}'}$.

In the next theorem, we present a lower bound for $\min_{\mathcal{MM} \cup \mathcal{AMM}'}$. Its proof is similar to the one used in the proof of Theorem 5.

Theorem 8 $\min_{\mathcal{MM} \cup \mathcal{AMM}'} \geq \min_{\mathcal{MM}} - MaxHEdge + 1$.

6. Related Work

Solutions to the committee coordination problem mostly focus on the three properties of the original problem described in Subsection 2.3 [2, 3, 4, 5, 6, 7]. In the seminal work by Chandy and Misra [2], the committee coordination problem is reduced to the dining or drinking philosophers problems [14]. Each philosopher represents a committee, neighboring philosophers have a common member, and a meeting is held only when the corresponding philosopher is eating. Bagrodia [3] solves the problem by introducing the notion of *managers*. Each manager handles a set of committees and two managers may have intersecting sets of assigned committees. Each committee member notifies its corresponding committee managers that it desires to participate. Conflicts between two committees (*i.e.*, committees that share a member) managed by the same manager are resolved locally within the manager. Conflicts between two committees managed by different managers are resolved using a circulating token. In a later work [4], Bagrodia combines a message count mechanism (to ensure Synchronization) with a reduction to dining/drinking philosophers (to ensure Exclusion).

Joung [19] extends the original committee coordination problem by considering fairness properties. One such property, called weak fairness in [19] or professor fairness in this paper, requires that if a professor is waiting to participate in some committee meeting, then he must eventually participate in a committee meeting (not necessarily the same). The main result is the impossibility of implementing a fair committee coordination algorithm if one of the following conditions hold:

- One process's readiness to participate in a committee can be known by another only through communication, and the time it takes two processes to communicate is not negligible.
- A process decides autonomously when it will attempt participating in a committee, and at a time that cannot be predicted in advance.

Joung's result holds for fairness on multi-party committees as well. Tsay and Bagrodia [5] reach the same result with respect to the second condition identified by Joung [19].

In [7], Kumar circumvents the impossibility result of Tsay and Bagrodia by making the following additional assumption: every professor waits for meetings infinitely often. In this model, Kumar proposes an algorithm that solves the committee coordination problem with professor fairness using multiple tokens, each representing one committee. Based on the same assumption, several other committee coordination algorithms that satisfy fairness can be found in [6].

7. Conclusion

In this paper, we proposed two Snap-stabilizing distributed algorithms for the committee coordination problem. The first algorithm satisfies 2-Phase Discussion as well as Maximal Concurrency. The second algorithm satisfies 2-Phase Discussion as well as Professor Fairness assuming that every professor waits for meetings infinitely often. As we showed, even under this latter assumption, satisfaction of both Maximal Concurrency and Professor Fairness is impossible.

For the second algorithm, we introduced and analyzed the degree of fair concurrency to show that it still allows high level of concurrency. We also evaluated an upper bound on waiting time. Finally, with a slight modification, we obtained another algorithm that respects Committee Fairness.

For future work, several interesting research directions are open. One can consider other combinations of properties. For instance, we conjecture that providing both Maximal Concurrency and bounded waiting time is impossible. Another problem is to design a fault-tolerant committee coordination algorithm in the message-passing model. An important issue is to address dynamic hypergraphs, where professors (processes) can enter or leave the hypergraph, and, new committees may be created or some committees may be dissolved or merged. Optimality is also an open question in that one can study the optimal bound on the degree of fair concurrency. Another interesting line of research is enforcing priorities on convening committees. Finally, we are planning to implement the algorithms presented in this paper in distributed code generation frameworks such as the one in [8]. Our algorithms will allow generating fully distributed code from high-level component-based models.

References

[1] B. Bonakdarpour, S. Devismes, F. Petit, Snap-stabilizing committee coordination, in: IPDPS'2011, 25th IEEE International Parallel and Distributed Processing Symposium, 2011, pp. 231–242.

[2] K. M. Chandy, J. Misra, Parallel program design: a foundation, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[3] R. Bagrodia, A distributed algorithm to implement n-party rendezvous, in: Foundations of Software Technology and Theoretical Computer Science, Seventh Conference (FSTTCS), 1987, pp. 138–152.

[4] R. Bagrodia, Process synchronization: Design and performance evaluation of distributed algorithms, IEEE Transactions on Software Engineering (TSE) 15 (9) (1989) 1053–1065.

[5] Y.-K. Tsay, R. Bagrodia, Some impossibility results in interprocess synchronization, Distributed Computing 6 (4) (1993) 221–231.

[6] C. Wu, G. Bochmann, M. Y. Yao, Fairness of n-party synchronization and its implementation in a distributed environment, in: Workshop on Distributed Algorithms (WDAG), 1993, pp. 279–293.

[7] D. Kumar, An implementation of n-party synchronization using tokens, in: Distributed Computung Systems (ICDCS), 1990, pp. 320–327.

[8] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, J. Sifakis, A framework for automated distributed implementation of component-based models, Distributed Computing 25 (5) (2012) 383–409.

[9] A. Bui, A. K. Datta, F. Petit, V. Villain, State-optimal snap-stabilizing pif in tree networks, in: A. Arora (Ed.), WSS, IEEE Computer Society, 1999, pp. 78–85.

[10] A. Bui, A. K. Datta, F. Petit, V. Villain, Snap-stabilization and PIF in tree networks, Distributed Computing 20 (1) (2007) 3–19.

[11] E. W. Dijkstra, Self-stabilizing systems in spite of distributed control, Communications of the ACM 17 (11).

[12] S. Dolev, A. Israeli, S. Moran, Uniform dynamic self-stabilizing leader election, IEEE Transactions on Parallel and Distributed Systems 8 (4) (1997) 424–440.

[13] S. Dolev, Self-stabilization, MIT Press, 2000.

[14] K. M. Chandy, J. Misra, The drinking philosophers problem, ACM Transactions on Programming Languages and Systems (TOPLAS) 6 (4) (1984) 632–646.

[15] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, J. Sifakis, Automated conflict-free distributed implementation of component-based models, in: IEEE Symposium on Industrial Embedded Systems (SIES), 2010, pp. 108–117.

[16] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, J. Sifakis, From high-level component-based models to distributed implementations, in: ACM International Conference on Embedded Software (EMSOFT), 2010, pp. 209–218.

[17] J. L. Welch, N. A. Lynch, A modular drinking philosophers algorithm, Distributed Computing 6 (4) (1993) 233–244.

[18] A. K. Datta, R. Hadid, V. Villain, A self-stabilizing token-based k-out-of-l exclusion algorithm, Concurrency and Computation: Practice and Experience 15 (11-12) (2003) 1069–1091.

[19] Y.-J. Joung, On fairness notions in distributed systems: I. a characterization of implementability, Information and Computation 166 (1) (2001) 1–34.

[20] M. Gairing, W. Goddard, S. T. Hedetniemi, P. Kristiansen, A. A. McRae, Distance-two information in self-stabilizing algorithms, Parallel Processing Letters 14 (3-4) (2004) 387–398.

[21] A. Arora, M. Gouda, Distributed reset, IEEE Transactions on Computers 43 (1994) 316–331.

[22] S. Dolev, T. Herman, Superstabilizing protocols for dynamic distributed systems, Chicago Journal of Theoretical Computer Science 1997.

[23] A. K. Datta, L. L. Larmore, P. Vemula, Self-stabilizing leader election in optimal space, in: Stabilization, Safety, and Security of Distributed Systems (SSS), 2008, pp. 109–123.

[24] S.-T. Huang, N.-S. Chen, Self-stabilizing depth-first token circulation on networks, Distributed Computing 7 (1) (1993) 61–66.

[25] A. K. Datta, C. Johnen, F. Petit, V. Villain, Self-stabilizing depth-first token circulation in arbitrary rooted networks, Distributed Computing 13 (4) (2000) 207–218.

[26] A. Cournier, S. Devismes, V. Villain, A snap-stabilizing DFS with a lower space requirement, in: Self-Stabilizing Systems (SSS), 2005, pp. 33–47.

[27] A. Cournier, S. Devismes, V. Villain, Light enabling snap-stabilization of fundamental protocols, ACM Transactions on Autonomous and Adaptive Systems (TAAS) 4 (1).