Snap-Stabilizing Committee Coordination

Borzoo Bonakdarpour*, Stéphane Devismes†, Franck Petit‡
*Department of Electrical and Computer Engineering
University of Waterloo
200 University Avenue West
Waterloo, Ontario, N2L 3G1, Canada
Email: borzoo@ece.uwaterloo.ca
†VERIMAG UMR 5104
Grenoble 1, France
Email: stephane.devismes@imag.fr
‡LIP6, UPMC Paris 6, France
Email: franck.petit@lip6.fr

***Abstract*—In this paper, we propose two *snap-stabilizing* distributed algorithms for the *committee coordination problem*. In this problem, a committee consists of a set of processes and committee meetings are synchronized, so that each process participates in at most one committee meeting at a time. Snap-stabilization is a versatile technique allowing to design algorithms that 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 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 Time*. From previous papers, we know that (1) in the general case, (weak) fairness cannot be achieved in the committee coordination, and (2) it becomes feasible provided that each process 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 time, and either maximal concurrency or fairness. The algorithm implementing fairness requires that every process 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.**

I. 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, accomplishing termination, reaching agreement, constructing 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 exposed to numerous types of faults.

The classic *committee coordination problem* [8] 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; it starts waiting and remains waiting until a meeting of a committee of which it 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 may convene 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.

A. Related Work

Solutions to the committee coordination problem mostly focus on the three aforementioned properties [2], [3], [8], [20], [22], [24]. In the seminal work by Chandy and Misra [8], the committee coordination problem is reduced to the dining or drinking philosophers problems [7]. Each philosopher represents a committee, neighboring philosophers have a common member, and a meeting is held only when the corresponding philosopher is eating. Bagrodia [2] 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 [3], 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 it 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 non-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 [22] reach the same result with respect to the second condition identified by Joung [19].

In [20], 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 [24].

B. Contributions

In this paper, we propose Snap-stabilizing distributed algorithms for the committee coordination problem. Snap-stabilization is a versatile technique allowing to design algorithms that 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 intervention.

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

We first consider Maximal Concurrency and Professor Fairness. As in [20], to circumvent the impossibility result of [22], each time we consider professor fairness in the sequel of the paper, we assume that every professor waits for meetings infinitely often. Under this assumption, we show that Maximal Concurrency and Professor Fairness are contradictory, *i.e.*, it is impossible to design a committee coordination algorithm (even non-stabilizing) that satisfies both simultaneously. 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 to participate 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 to hold concurrently. That is, making maximal concurrency impossible.

Consequently, we focus on the aforementioned contradictory properties independently by providing two algorithms.

The former maximizes concurrency at the cost of not ensuring professor fairness. By contrast, the latter 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 (s)he can participate in a meeting. Such a mechanism can be implemented using a token circulating among the professors. To ensure fairness, when a process holds a token, it has the higher priority to convene a meeting and it retains the token until it joins the meeting. In that case, some neighbors of the token holder can be prevented to participate in other meetings so that the token holder eventually does, resulting in weakening the concurrency. In order to guarantee maximal concurrency (but at the risk of being unfair), a waiting process must release the token if it is not yet able to convene a meeting to give a chance to other committees in which all members are already waiting.

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 Time.

We also propose a Snap-stabilizing algorithm that satisfies Fairness on professors (respectively, committees) and respects 2-Phase Discussion Time, assuming that every professor waits for meetings 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 II, we present the preliminary concepts. Section III is dedicated to definitions of Maximal Concurrency and Fairness in committee coordination. Then, in Section IV, we introduce 2-phase Discussion Time and our first snap-stabilizing algorithm that satisfies both Maximal Concurrency and 2-phase Discussion Time. In Section V, we propose our snap-stabilizing algorithm that satisfies Fairness and 2-phase Discussion Time. Our analysis on level of concurrency and waiting time is also presented in this section. Finally, we present concluding remarks and discuss future work in Section VI.

Due to reasons of space all proofs have been omitted. All detailed proof are available in the technical report inline at http://www.ece.uwaterloo.ca/~bbonakda/ipdps11.pdf.

II. Background

A. 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. For the sake of simplicity, we assume that each committee has at least two members (adapting our results to take singleton committees into account is straightforward). Hence, we model a *distributed system* as a simple self-loopless *static* 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 $\epsilon \subseteq 2^V$ (*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 a vertex 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 the distributed system $\mathcal{H} = (V, \mathcal{E})$ is an undirected simple connected graph $G_{\mathcal{H}} = (V, E_{\mathcal{E}})$, where $E_{\mathcal{E}} = \{\{p_1, p_2\} \mid p_1 \in V \wedge p_2 \in V \wedge p_1 \in N(p_2)\}$. Note that we use hypergraphs only because they present the concepts of the committee coordination problem elegantly and simplify reasoning about correctness of our results. However, it is straightforward to see that one can model the distributed system using the underlying communication network.

B. 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. The *state* of a process is defined by the value of its variables. A process can change its state by executing its *local algorithm*. The local algorithm of a process is described using a finite set 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 at process 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.

A *configuration* γ of 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.

A *computation* of a distributed systems is a maximal sequence of configurations γ_0, γ_1, ... such that γ_0 is an arbitrary configuration and for each configuration γ_i, $i \geq 0$, the next configuration γ_{i+1} is obtained by *atomically* executing the statement of at least one action that is enabled in γ_i. *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).

Computations are driven by a *daemon* (or *scheduler*). A daemon is defined as a predicate on 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 chooses at least one (maybe more) of these processes to execute an action. *Weak fairness* means that every continuously enabled process eventually executes an enabled action.

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* [16]. This notion captures the execution rate of the slowest process in any execution. The first *round* of an execution e is the minimal prefix of e, $\gamma_0 \ldots \gamma_i$, 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 *parallel composition* (or simply *composition*) [21] of two algorithms $\mathcal{P}_1$ and $\mathcal{P}_2$ is denoted by $\mathcal{P}_1 \circ \mathcal{P}_2$ and is the union of guarded actions of $\mathcal{P}_1$ and $\mathcal{P}_2$. Thus, computations of $\mathcal{P}_1 \circ \mathcal{P}_2$ is obtained by considering all possible interleavings of $\mathcal{P}_1$ and $\mathcal{P}_2$.

C. Snap-stabilization

Snap-stabilization [6] is a versatile technique that allows designing algorithms that 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 intervention. In contrast, the related concept of self-stabilization [14] only guarantees that the system *eventually* recovers to a correct behavior.

In (self- or snap-) stabilizing systems, we consider the system immediately after the end 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. A snap-stabilizing algorithm then guarantees that *starting from any initial configurations, any of its computations always satisfies the specification of the problem at hand*.

It is important to note that a snap-stabilizing algorithm is not insensitive to transient faults. In fact, a snap-stabilizing algorithm guarantees that any task execution started after the end of the faults will operate correctly. However, we have no guarantees for tasks executed completely or in part during faults. For example, in the committee coordination problem, every meeting convened after the last transient faults will satisfy every requirement of the committee coordination problem. However, we have no guarantee for the meetings started during the transient faults, except that they will not interfere with the execution of the other meetings.

D. The Committee Coordination Problem

The *original committee coordination problem* is as follows [8]. 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 one 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 it is a member of, agree on meeting. Professors may leave a meeting, become idle, and subsequently be waiting for a new committee meeting.

Chandy, Misra [8], and Bagrodia [3] 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 be started 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.

III. Maximal Concurrency versus Fairness in Committee Coordination

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 [8] and Bagrodia [2], [3] result in decreasing the level of concurrency drastically, making them less appealing for practical purposes. Examples include the circulating token mechanism among conflicting committees [2], and reduction to the dining philosophers problems, where a "*manager*" handles multiple committees. Reduction to the drinking philosophers problem such as those in [3], [8], [23] 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 [11]. 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. Now, to formally define this constraint, we let a professor (process) remain 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 1 (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 starts.*

As mentioned in Subsection I-A, if a professor's status does not become waiting infinitely often, achieving fairness is impossible [22]. 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) as follows.

Definition 2 (Professor Fairness) *Every professor eventually participates in a committee meeting that it is a member of.*

The next theorem shows that Maximal Concurrency and Professor Fairness are imcompatible. Its proof follows ideas similar to the impossibility results of Joung [19] as well as Tsay and Bagrodia [22].

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

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 3 (Committee Fairness) *Every committee meeting eventually convenes.*

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 for an arbitrary distributed system that solves the committee coordination problem and simultaneously satisfies Maximal Concurrency and Committee Fairness even in a non-stabilizing context.*

Theorem 1 clearly 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 IV. Omitting maximal concurrency results an algorithm such as the one presented in Section V.

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 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 [17]) instead of a single one would enhance the fairness guarantee. However, increasing the number of tokens results in decreasing the degree of (fair) concurrency, which is the target metric here. The key idea is that the token is used to give the highest 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 starts 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 are 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, enhancing the fairness makes the concurrency decreasing: fairness and concurrency are orthogonal properties in the committee coordination problem.

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

In this section, our goal is to develop a Snap-stabilizing algorithm that satisfies Maximal Concurrency as well as a stronger synchronization property called 2-Phase Discussion Time. This latter property is defined and justified in Subsection IV-A. We present our algorithm in Subsection IV-B.

A. 2-Phase Discussion Time

The original 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 liberally with respect to this issue. For instance, solutions proposed in [3], [8] that employ the dining philosophers problem [7] in order to resolve committee conflicts satisfy the above specification, 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[1]. There are two problems with such a restriction: (1) 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 starts. 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 [4], [5]. The following definition elegantly captures this requirement.

Definition 4 (2-Phase Discussion Time) *We define the 2-phase discussion time by the following two properties:*

- `Phase 1.` *(Essential Discussion Time) Upon a meeting starts, each participating professor must remain in the meeting for some finite time period.*
- `Phase 2.` *(Voluntary Discussion Time) Upon a meeting starts and after fulfilling the essential discussion, the discussion (and consequently the meeting) continues for a (possibly zero) finite time 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 discussion times.

B. Algorithm

Our algorithm is the parallel 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 that manages a circulating token – denoted $\mathcal{TC}$ – for ensuring Progress.

[1]The same argument holds for solutions based on the *drinking philosophers* [7] and tokens.

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 [1], [13], [15]) with one of the self-stabilizing token circulation algorithms in [9], [10], [12], [18] 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. We note that the composition $\mathcal{CC}1 \circ \mathcal{TC}$ does not explicitly contain Action T: In the composite algorithm, 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: The Algorithm $\mathcal{CC}1$ (see Algorithm 1) is identical for all processes in the distributed system. 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.

Interactions between each professor p and its local algorithm are managed using two input predicates: $RequestIn(p)$ and $RequestOut(p)$. These predicates materializes the fact that each professor autonomously decides to wait and leave a meeting. The predicate $RequestIn(p)$ holds when professor p requests its participation in a committee meeting. After a committee convenes, the predicate $RequestOut(p)$ holds when p fulfills its essential discussion and voluntarily stops discussing. Thus, since p has done its essential discussion, p eventually satisfies $RequestOut(p)$. Once $RequestOut(p)$ is true, it remains true until p becomes idle.

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

- 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 $Step_1$).
- 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 $Step_{22}$). We call the set of such hyperedges $FreeEdges_p$.
  To obtain agreement on the committees to convene, we implement token-based priorities as follows. 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), the looking process with the maximum identifier breaks the tie. 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 either the token holder is currently in a meeting or 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.
- Once all processes of a hyperedge are looking and they agree on that hyperedge, they are all ready to start their discussion. To this end, a process changes its status from looking to waiting[2] to show that it is waiting for the committee to convene (Action $Step_{31}$). The committee convenes since all the members have changed their status to waiting. Then, each process executes its essential discussion and switches its status to done (Action $Step_{32}$).
- Finally, a process is allowed to leave the committee meeting when all processes of the committee have fulfilled their essential discussion (*i.e.*, they are all in

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

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:

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

Predicates:

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

Actions:

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

the done status). Then, the meeting takes place until a process p unilaterally decides to leave it (that is, until $RequestOut(p)$ is $true$) after finite period of voluntary discussion time. To leave the committee, it switches its status to idle again, resets its hyperedge pointer, and releases the token if it owns it (Action $Step_4$).

The rest of actions of the algorithm deal with token circulation and snap-stabilization. In particular, action $Token_1$ deals with acquiring the token and 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, or, there does not exist an available committee for p to participate, then it releases the token (action $Token_2$). Finally, actions $Stab_1$ and $Stab_2$ correct the state of a process, if faults perturb the state of a 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 and enabled to reset to idle.

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.*

V. 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 the sake of 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 code of the next algorithm. In Subsection V-A, we present a snap-stabilizing algorithm that guarantees the properties of 2-phase committee coordination and Professor Fairness. Then, in Subsection V-B, we present the notion of *degree of fair concurrency* to measure concurrency while preserving Professor Fairness and analyze our algorithm presented in Subsection V-A with respect to this measure. We also compute the worst case waiting time of our algorithm in this Subsection. Finally, we discuss Committee Fairness in Subsection V-C.

A. Algorithm

Our algorithm is the parallel composition of two modules: (1) a Snap-stabilizing algorithm – denoted $\mathcal{CC}2$ – that ensures Exclusion, Synchronization, and 2-Phase Discussion, and (2) a self-stabilizing module that manages a circulating token – denoted $\mathcal{TC}$ – for ensuring Fairness and consequently Progress. (Remark 1 holds for this composition as well.) Algorithm $\mathcal{CC}2$ (see Algorithm 2) is identical for all processes in the distributed system and we assume that each process has a unique identifier and the set of all identifiers is a total order. Note also that Action T is emulated in Algorithm $\mathcal{CC}2$ in the same way as in Algorithm $\mathcal{CC}1$.

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), all processes not in ϵ must not wait for processes involved in the committee ϵ. To that goal, we introduce the Boolean variable L, which shows whether or not the 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 $\mathcal{CC}1$.

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.*

B. Complexity Analysis

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 the meeting for infinite time.*[3] *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.*

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* of hypergraph $\mathcal{H} = (V, \mathcal{E})$ is a subset S of hyperedges of $\mathcal{H}$, such that no two hyperedges

[3] As in Definition 1, infinite meetings are used only for formalization.

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

Macros:

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

Predicates:

$$
\begin{array}{lcl}
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 \ (\forall q \in \epsilon \ : \ (P_q = \epsilon \ \Rightarrow \ S_q = \mathsf{done}))) \\
LocalMax(p) & \equiv & p = \max(FreeNodes_p) \\
MaxToFreeEdge(p) & \equiv & \neg Token(p) \ \wedge \ \neg T_p \ \wedge \ \neg Locked(p) \ \wedge \ \neg L_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 T_p \ \wedge \ \neg Locked(p) \ \wedge \ \neg L_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 \ T_p \ \wedge \ (S_p = \mathsf{looking}) \ \wedge \ (P_p \notin MinEdges_p) \\
JoinTokenHolder(p) & \equiv & \neg Token(p) \ \wedge \ \neg T_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 \\
 & & \ [\exists \epsilon \in \mathcal{E}_p \ : \ (P_p = \epsilon \ \wedge \ ((\forall q \in \epsilon \ : \ (P_q = \epsilon \ \wedge \ S_q \in \{\mathsf{looking}, \mathsf{waiting}\})) \vee \\
 & & \ (\forall q \in \epsilon \ : \ (P_q = \epsilon \ \wedge \ S_q \in \{\mathsf{waiting}, \mathsf{done}\}))))]] \ \wedge \\
 & & [(S_p = \mathsf{done}) \ \Rightarrow \\
 & & \ [\exists \epsilon \in \mathcal{E}_p \ : \ (P_p = \epsilon \ \wedge \ ((\forall q \in \epsilon \ : \ (P_q = \epsilon \ \wedge \ S_q \in \{\mathsf{waiting}, \mathsf{done}\})) \vee \\
 & & \ (\forall q \in \epsilon \ : \ (P_q = \epsilon \ \Rightarrow \ S_q = \mathsf{done}))))]]
\end{array}
$$

Actions:

$Step_{11}$	::	$TokenHolderToEdge(p)$	$\mapsto$	$P_p := \epsilon$ 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 $P_{\max(FreeNodes_p)} = \epsilon$;
$Step_2$	::	$Ready(p) \wedge (S_p = \mathsf{looking})$	$\mapsto$	$S_p := \mathsf{waiting}$;
$Step_3$	::	$Meeting(p) \wedge (S_p = \mathsf{waiting})$	$\mapsto$	$\langle \mathsf{EssentialDiscussion} \rangle$, $S_p := \mathsf{done}$;
$Step_4$	::	$LeaveMeeting(p) \wedge RequestOut(p)$	$\mapsto$	$S_p := \mathsf{looking}$, $P_p :=\perp$, **if** $Token(p)$ **then** $ReleaseToken_p$ **fi**, $T_p := false$;
$Token$	::	$Token(p) \neq T_p$	$\mapsto$	$T_p := Token(p)$;
$Lock$	::	$Locked(p) \neq L_p$	$\mapsto$	$L_p := Locked(p)$;
$Stab$	::	$\neg Correct(p)$	$\mapsto$	$S_p := \mathsf{looking}$, $P_p :=\perp$;

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 that 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, $\mathcal{MM} \subseteq \mathcal{M}$. Note 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, noted $|\epsilon|$, 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 $\{m \in \mathcal{MM}_{\mathcal{H}_{\overline{X}}} \mid \forall q \in \epsilon \backslash X \ : \ q$ 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 the 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.

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

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

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

Since algorithm $\mathcal{CC}2 \circ \mathcal{TC}$ satisfies Professor Fairness, it is of practical interest to evaluate its 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.*

To evaluate Waiting Time, we need to introduce $\max_{Disc}$ which is the maximum amount of time a process discusses in a meeting. We assume that $\mathcal{TC}$ is a fair composition of the token circulation algorithm in [10] and the leader election algorithm in [13]. 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 $O(\max_{Disc} \times n)$, where n is the number of processes.*

C. 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 $MaxHEdge = \max_{\epsilon \in \mathcal{E}} |\epsilon|$.

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

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

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

VI. Conclusion

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

REFERENCES

[1] A. Arora and M. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43:316–331, 1994.

[2] R. Bagrodia. A distributed algorithm to implement n-party rendevouz. In *Foundations of Software Technology and Theoretical Computer Science, Seventh Conference (FSTTCS)*, pages 138–152, 1987.

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

[4] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis. Automated conflict-free distributed implementation of component-based models. In *IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 108–117, 2010.

[5] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis. From high-level component-based models to distributed implementations. In *ACM International Conference on Embedded Software (EMSOFT)*, pages 209–218, 2010.

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

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

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

[9] A. Cournier, S. Devismes, and V. Villain. A snap-stabilizing DFS with a lower space requirement. In *Self-Stabilizing Systems (SSS)*, pages 33–47, 2005.

[10] A. Cournier, S. Devismes, and V. Villain. Light enabling snap-stabilization of fundamental protocols. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(1), 2009.

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

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

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

[14] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.

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

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

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

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

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

[20] D. Kumar. An implementation of n-party synchronization using tokens. In *Distributed Computung Systems (ICDCS)*, pages 320–327, 1990.

[21] G. Tel. *Introduction to distributed algorithms*. Cambridge University Press, Cambridge, UK, Second edition 2001.

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

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

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