Efficient Implementation of Synchronous Communication Over Asynchronous Networks

Manhoi Choy and *Ambuj K. Singh*

Department of Computer Science
University of California at Santa Barbara
Santa Barbara, California 93106

Abstract

The implementation of synchronous communication primitives over asynchronous message-passing networks is investigated. Efficient algorithms are presented for the committee coordination problem and its more dynamic version called the first-order interaction problem. A new notion of strong fairness called 1-sf and algorithms achieving it are described. It is further shown that any stronger notion of fairness cannot be implemented.

Keywords: Concurrency, Distributed Algorithms, Fairness, Synchronization.

1 Introduction

Design and implementation of efficient inter-process communication primitives is an important goal for parallel and distributed systems. This paper investigates the case of *synchronous* communication primitives. These primitives occur in languages such as Ada [17] and CSP [5], and programming paradigms such as *Shared Actions* [20], *Joint Actions* [3], *Raddle* [10], IP [12], and *Script* [13]. The basic form of synchronous communication, supported by Ada and CSP, allows process synchronization only between two processes. This kind of primitive has been later generalized to allow communication and synchronization between an arbitrary number of processes [3, 10, 20]. This general setting has been abstracted by Chandy and Misra [6] as the Committee Coordination problem: each professor in a university serves on one or more committees, each committee has a fixed set of members who may be waiting to convene a committee from time to time, and a committee cannot convene until all its members are present. A solution to the problem should ensure that committees with common members are not scheduled at the same time and that a committee that is continuously ready is eventually scheduled.

The *first-order interaction* problem [14] is a variation of the committee coordination problem that provides some degree of dynamism. In contrast to committees (henceforth called zeroth-order interactions) where the participants are fixed in advance, first-order interactions support dynamically varying sets of participants. A first-order interaction $\mathcal{I}$ is represented by a set of *roles* $R(\mathcal{I})$. Processes become *ready* to enrole into some specific roles of some interactions. An interaction can be committed only if all its roles are fulfilled. Provided sufficiently many disjoint sets of processes are enroled into roles of $R(\mathcal{I})$, many instances of $\mathcal{I}$ can be committed simultaneously. First-order interactions allow more concurrency than their zeroth-order counterparts and are also inherently more fault tolerant. Existing constructs that are first-order include Script and the *team* construct of Raddle and IP. In this paper, we investigate the implementation of zeroth-order and first-order interactions over asynchronous message-passing networks. We present a simple and efficient algorithm that solves both the versions of the problem with a better response time than existing solutions.

The progress requirement in both the zeroth-order and the first-order interaction problem is one of weak fairness: any continuously ready interaction should be committed. Stronger notions of fairness have been investigated for these problems [2, 11, 24]. Existing classification of fairness notions for interactions is along two orthogonal directions: the subject of fairness and the level of fairness [2, 11]. A subject may be a process, a group of processes, or an interaction. The level of fairness may be weak, strong, and unconditional. This gives us a combination of nine types of fairness notions: *weak process fairness, weak group fairness, weak interaction fairness, strong process fairness*, etc. Though a number of algorithms achieving weak fairness conditions have been proposed, only a few algorithms exist for strong fairness. Furthermore, there exist impossibility results [24] showing that some kinds of strong fairness are not implementable. We investigate the implementability of various strong fairness notions and propose a new notion of strong fairness called 1-sf.

For the case of binary interactions (every interaction consists of exactly two processes), this new notion of fairness is equivalent to strong fairness with respect to processes (*SPF*). For the case of general interactions, this notion of fairness is stronger than weak fairness with respect to interactions (*WIF*) and weaker than *SPF*. We present algorithms achieving 1-sf for binary and general interactions and show that any stronger notion of fairness (i.e., k-sf for $k \geq 2$) cannot be implemented.

The rest of the paper is organized as follows: In Section 2, we consider the zeroth-order and the first-order interaction problems. In Section 3, we investigate the problem of implementing strong fairness and define and implement 1-sf. We conclude with a brief discussion in Section 4.

2 Zeroth-order and First-order Interactions

2.1 Problem Statement

We first discuss the statement of the zeroth-order interaction problem. This problem consists of a set of processes and a set of interactions. Each process p has a predetermined set of interactions $I(p)$ that it may participate in and each interaction $\mathcal{I}$ has a predetermined set of processes $P(\mathcal{I})$ that co-operate to execute it. Processes that may participate in the same interaction are physically close to each other and can communicate with each other directly over a message-passing network. Two interactions $\mathcal{I}$ and $\mathcal{J}$ *conflict* with each other if they share a common process, i.e., if $P(\mathcal{I}) \cap P(\mathcal{J}) \neq \{\}$. The graph induced by the conflict relation is called the *conflict graph*. A process can be in one of three states: *idle*, *ready*, and *commit*. Initially every process is idle. At an undetermined time, an idle process p becomes ready to take part in interactions $I(p)$. An interaction $\mathcal{I}$ is said to be *ready* if all processes in $P(\mathcal{I})$ are ready. A ready interaction can be later *committed*, at which time all processes in $P(\mathcal{I})$ also become committed. The commitment of each interaction $\mathcal{I}$ eventually terminates, at which point the processes in $P(\mathcal{I})$ become idle again. A ready interaction can also become non-ready without being committed on account of some participating processes committing to another interaction.

A solution to the zeroth-order problem should satisfy two safety requirements, *mutual exclusion* and *synchronization*, and one progress requirement, *WIF*. Mutual exclusion requires that a process does not participate in two different interactions simultaneously. Synchronization requires that the commitment of an interaction is only started if it is ready. Weak interaction fairness requires that some process in a continuously ready interaction is committed eventually.

The zeroth-order interaction problem outlined above is *static* in the sense that interactions have fixed sets of processes and processes have fixed sets of interactions in which they participate. A more dynamic abstraction of the problem called the first-order interaction problem allows the relationship between interactions and processes to be captured through an intermediate set of entities called *roles*. In this problem, each interaction $\mathcal{I}$ has a fixed set of *roles* $R(\mathcal{I})$ into which processes enrole. Each role $r \in R(\mathcal{I})$ has a fixed set $P(r)$ of processes that can enrole in it. As before, a process can be in one of three states: idle, ready, or commit. An idle process becomes ready to fulfill some specific roles of a set of interactions. On account of the possibility of more than one process satisfying a role, more than one instance of an interaction may be committed at any time. An interaction $\mathcal{I}$ is said to be *ready* if all its roles can be fulfilled by ready processes. An instance of a ready interaction can be later *committed*, at which time all processes assigned to the roles in the instance also become committed. The commitment of each instance eventually terminates, at which point all the processes participating in the instance become idle again.

Analogous to the zeroth-order interaction problem, a solution to the first-order interaction problem must satisfy the requirements of *mutual exclusion*, *synchronization*, and *weak interaction fairness*. The requirements of mutual exclusion and synchronization are similar to the case of zeroth-order interactions. Weak interaction fairness requires that if a set of processes that can fulfill all the roles of an interaction is continuously ready then an instance of the interaction consisting of some processes in the set will be committed eventually.

2.2 An Algorithm

We use a technique due to Chandy and Misra [6] and solve the first-order interaction problem by transforming the problem to the *dining philosophers problem* [16]. The zeroth-order interaction problem is also solved as a result since it is a restriction of the first-order interaction problem. To begin, we give a brief account of the dining philosophers problem.

The dining philosophers problem consists of a set of processes and a set of resources modeled by a graph, with each vertex representing a process and each edge representing a resource shared by the end vertices. A process may be in one of three states: *thinking*, *hungry*, and *eating*. Initially, a process is thinking but it can become hungry at any time. A hungry process starts eating when it is granted exclusive access to the set of neighboring resources. The resources are released eventually as all eating periods are required to be finite. A process then transits back to the thinking state. Neighboring processes *conflict* with each other as they may require the same resource at the same time. A solution to the dining philosophers problem is required to synchronize these processes so that *mutual exclusion* and *starvation freedom* are satisfied. Mutual exclusion requires that neighboring philosophers do not eat simultaneously. Starvation freedom requires that a hungry philosopher eats eventually.

A high level description of our algorithm, called Algorithm 1, for the first-order interaction problem is shown in figures 1 and 2. The algorithm is specified by a set of events: the predicate in the left column defines the condition under which an event is enabled and the corresponding code in the right column describes the action due to the event. For each interaction $\mathcal{I}$, we designate a process in $P(\mathcal{I})$ as the coordinator of the interaction. This process is denoted c_I. When a process p is ready to enrole into some roles of $\mathcal{I}$, it sends a $\langle ready \rangle$ message, together with an identification of the roles it wishes to enrole in, to coordinator c_I. (For simplicity, the identification of the roles is not shown in the figure.) Coordinator c_I has a philosopher f_I associated with it. When c_I receives $\langle ready \rangle$ messages from a number of processes sufficient to fulfill all the roles of $\mathcal{I}$, philosopher f_I becomes hungry. (Set $ReadySet$ stores the processes that have sent a ready message to the coordinator.) The philosophers execute a solution to the dining philosophers problem. Two philosophers conflict with each other if the corresponding interactions conflict. When philosopher f_I is allowed to eat by the underlying dining philosophers algorithm, coordinator c_I sends $\langle request \rangle$ message to each of the processes from whom it has received a $\langle ready \rangle$ signal to check if they are still ready for the roles of $\mathcal{I}$. Processes reply with a $\langle yes \rangle$ message if they are ready, and a $\langle no \rangle$ message otherwise. From the processes that replied with a $\langle yes \rangle$ message, c_I activates as many instances of $\mathcal{I}$ as possible by sending $\langle commit \rangle$ messages to processes. (These $\langle commit \rangle$ messages also identify the interaction instance and the role that a process has been designated to fulfill. For simplicity, this additional information is not shown in the figures.) Any remaining processes are released by sending $\langle no\text{-}commit \rangle$ messages. Processes respond to $\langle commit \rangle$ and $\langle no\text{-}commit \rangle$ messages by sending acknowledgements. When coordinator c_I receives all the acknowledgements, it sets philosopher f_I to the thinking state.

Condition	Action
idle to *ready* transition	$\langle \forall \mathcal{I} \ : \langle \exists r : r \in R(\mathcal{I}) \ \wedge \ p$ is ready for role $r \rangle$: **send** $\langle ready \rangle$ to $c_I \rangle$
Receive $\langle request \rangle$ from c_I	**if** *ready* **then send** $\langle yes \rangle$ to c_I **else send** $\langle no \rangle$ to c_I
Receive $\langle commit \rangle$ from c_I	**send** $\langle ack \rangle$ to c_I; commit to interaction $\mathcal{I}$
Receive $\langle no\text{-}commit \rangle$ from c_I	**send** $\langle ack \rangle$ to c_I

Figure 1: Algorithm 1 for process p

2.3 Correctness and Performance Evaluation

We introduce some terminology in order to discuss the correctness and the performance of Algorithm 1. For the interaction problems (zeroth and first-order), define response time to be the maximum period of time for which a set of processes fulfilling all the roles of an interaction remains ready and message complexity as the amount of information (in terms of the number of messages) needed for a ready interaction to transit to either the commit or the idle state. For the dining philosophers problem, define response time as the maximum amount of time for which a philosopher remains hungry and message complexity as the amount of information sent per eating period.

To measure time in a asynchronous system we assume ν to be the maximum message delivery time between neighboring processes and τ to be the maximum eating period in the dining philosophers problem. We also assume that local processing times are negligible as compared to ν and τ. Since the underlying dining philosophers algorithm requires that eating periods be finite, we first prove the following lemma.

Lemma 1 *The eating period of a philosopher is finite and at most* 4ν *in Algorithm 1.*

Receive $\langle ready \rangle$ from p	**insert** p into $ReadySet$
$ReadySet$ contains enough processes to execute $\mathcal{I}$	**set** f_I to *hungry*; **wait** until f_I is *eating*; **send** $\langle request \rangle$ to each process in $ReadySet$; **wait** until each process replies; remove process that replied $\langle no \rangle$ from $ReadySet$; **activate** as many instances of $\mathcal{I}$ as possible from those processes that have replied $\langle yes \rangle$ by sending $\langle commit \rangle$ messages and remove these processes from $ReadySet$; **send** $\langle no\text{-}commit \rangle$ to the remaining processes that have replied with $\langle yes \rangle$ messages; **wait** for $\langle ack \rangle$ from each $\langle commit \rangle$ or $\langle no\text{-}commit \rangle$ message; **set** f_I to *thinking*

Figure 2: Algorithm 1 for coordinator c_I

Proof: According to the algorithm in Figure 2, once a philosopher starts eating, the corresponding coordinator sends $\langle request \rangle$ messages to processes, waits until either $\langle yes \rangle$ or $\langle no \rangle$ messages are received, sends $\langle commit \rangle$ or $\langle no\text{-}commit \rangle$ messages, and waits for the $\langle ack \rangle$ messages. Since processes respond immediately to $\langle request \rangle$, $\langle commit \rangle$, and $\langle no\text{-}commit \rangle$ messages, the maximum duration of the above sequence of message exchanges is at most 4ν. □

Since eating periods are finite, by the correctness of the underlying dining philosophers algorithm, the philosophers satisfy the mutual exclusion and starvation freedom properties. The correctness of Algorithm 1 can be shown based on these properties.

Theorem 1 *Algorithm 1 satisfies the requirements of mutual exclusion, synchronization, and* WIF.

Proof: Consider the mutual exclusion requirement first. We have to show that a process is committed to at most one instance of at most one interaction. From the correctness of the underlying dining philosophers algorithm and our definition of the conflict relationship, no two philosophers eat at the same time if the associated interactions share a common process. Since, coordinators check the readiness of participating processes only while the associated philosophers are eating, coordinators of conflicting interactions do not overlap with each other in checking the readiness of participating processes. Therefore, a ready process will have at most one outstanding $\langle yes \rangle$ message and consequently will receive at most one $\langle commit \rangle$ message. Once a process is committed, it responds with a $\langle no \rangle$ message to all requests. Hence, mutual exclusion is satisfied.

Next, we consider the synchronization requirement. We have to show that an interaction is committed only if it is ready (or, equivalently, only if all its roles can be fulfilled by ready processes). According to the algorithm, an interaction is committed only after the corresponding philosopher has started eating and the coordinator has ascertained that there are enough ready processes to fulfill the roles of the interaction. So, the only way the synchronization requirement can be violated is if a ready process becomes non-ready while the coordinator is checking the states of the other processes. But a ready process can become non-ready only if some neighboring interaction is committed in the meantime. This cannot happen because an interaction is committed only while the corresponding philosopher is eating and because from the mutual exclusion property of the underlying solution, neighboring philosophers do not eat at the same time.

Finally, we consider the weak interaction fairness requirement. Consider a set of continuously ready processes Q that can fulfill all the roles of an interaction $\mathcal{I}$. We have to show that an instance of interaction $\mathcal{I}$ consisting of some processes in Q will be committed eventually. According to the algorithm, processes in Q will eventually send $\langle ready \rangle$ messages to coordinator c_I. Therefore, eventually $Q \subseteq ReadySet$ at coordinator c_I. Consequently, philosopher f_I will become hungry. From the correctness of the underlying dining philosophers algorithm, philosopher f_I will eventually eat. At this time, coordinator c_I will check the readiness of the processes in $ReadySet$ and commit an instance of $\mathcal{I}$ consisting of some processes in Q. $\square$

Let δ be degree of the conflict graph of the philosophers resulting from the transformation in Algorithm 1.

Theorem 2 *For any f and g, if the response time of the underlying dining philosophers algorithm is $f(\delta)*\tau+g(\delta)*\nu$, then the response time of Algorithm 1 is $4f(\delta)*\nu+g(\delta)*\nu+4\nu$. Also, if the message complexity of the underlying dining philosophers algorithm is $M(\delta)$ then the message complexity of Algorithm 1 is $M(\delta)+5m$ where $m = max_{\mathcal{I}}|P(\mathcal{I})|$.*

Proof: We analyze the maximum amount of time for which a set of processes Q fulfilling all the roles of an interaction $\mathcal{I}$ remains ready. According to the algorithm, processes in Q will send a $\langle ready \rangle$ message to coordinator c_I. These messages will be received within ν time units and coordinator c_I will set philosopher f_I to hungry. By Lemma 1, $\tau \leq 4\nu$. Therefore, philosopher f_I will start eating within $4f(\delta)*\nu+g(\delta)*\nu$ time units. Then, coordinator c_I will send a $\langle request \rangle$ message to processes in Q. These messages will be received within ν time units. Subsequently, processes in Q will respond with either a $\langle yes \rangle$ or a $\langle no \rangle$ message. These messages will be received at c_I within ν time units. In case some process sends a $\langle no \rangle$ message, the response time is bounded by $4f(\delta)*\nu+g(\delta)*\nu+2\nu$ time units. If all the processes respond with a $\langle yes \rangle$ message then c_I will send a $\langle commit \rangle$ message to the processes and commit an instance of the interaction. The response time in this case is bounded by $4f(\delta)*\nu+g(\delta)*\nu+4\nu$ time units.

The message complexity of $M(\delta)+5m$ follows as $M(\delta)$ messages are sent in the underlying dining philosophers algorithm and $5m$ messages including at most m $\langle ready \rangle$ messages, at most m $\langle request \rangle$ messages, at most m $\langle yes \rangle$ or $\langle no \rangle$ messages, at most m $\langle commit \rangle$ or $\langle no\text{-}commit \rangle$ messages, and at most m $\langle ack \rangle$ messages are sent per eating session. (Note that at the end of an eating session of a philosopher, for any set of processes $Q \subseteq Readyset$

that fulfills all the roles of the corresponding interaction, some process in Q becomes non-ready.) $\square$

In order to compare our algorithms to existing solutions, we choose a dining philosophers algorithms presented by us in [7]. For this algorithm, $f(\delta) = g(\delta) = O(\delta^2)$ and $M(\delta) = O(\delta)$. From the transformation used in Algorithm 1, $\delta \leq km$, where $k = max_p|I(p)|$ and m is as defined earlier. Consequently, the resulting solution to the first-order interaction problem has a response time of $O(k^2m^2\nu)$ and a message complexity of $O(km)$. The algorithm by Joung and Smolka [14] has a response time of $O(N\nu)$, where N is the total number of interactions in the system, and the number of messages sent is $6m$ per process per interaction. The response time of our algorithm is considerably better as it depends only on local parameters instead of N. The message complexities are incomparable as more than one instance of an interaction can be committed by a single coordinator in our algorithm. As opposed to our algorithm where the coordinators are fixed in advance, the algorithm by Joung and Smolka [14] chooses coordinators dynamically from the set of ready processes. This improves the fault-tolerance of their algorithm. Fault-tolerance of our algorithm can be improved by choosing a suitable fault-tolerant dining philosophers algorithm [7].

Restrictions	Authors/Algorithms	Response time	Message Complexity
$m = 2$	Buckley and Silberschatz [5]	$O(N\nu)$	$O(k)$
	Sistla [21]	$O(k^3(\tau_c + \nu))$	$O(k)$
	Bagrodia and Tsay [4]	$O(k^2\nu)$	$O(k)$
	Tsay and Bagrodia [23]	$O(k^2\nu)$	$O(k)$
	Our algorithm	$O(k^2\nu)$	$O(k)$
Arbitrary m	Kumar [15]	$O(N\nu)$	$O(km)$
	Park and Kim [18]	$O(N\nu)$	$O(km)$
	Ramesh [19]	$O(N\nu)$	$O(km)$
	Our algorithm	$O(k^2m^2\nu)$	$O(km)$

Figure 3: Comparison with existing algorithms to the zeroth-order interaction problem. Here, m is the number of processes required to execute an interaction, k is the number of interactions in which a process may participate, N is the total number of interactions in the system, and τ_c is the maximum duration of commitments.

The first-order interaction problem reduces to the zeroth-order interaction problem when each role is satisfied by exactly one process and no process satisfies more than one role per interaction. Figure 3 compares our algorithm with existing algorithms. For the case of binary interactions, i.e., $m = 2$, our algorithm achieves the same response time and message complexity as the best known solutions. For the case of general interactions, our algorithm is the only one whose response time is independent of N, the total number of interactions.

3 Process Synchronization under Strong Fairness

In this section we investigate strong fairness for the zeroth-order interaction problem. As mentioned earlier, strong fairness can be defined with respect to a process, a group, or an interaction:

- *Strong fairness with respect to processes (SPF):* A process that is ready to participate in ready interactions infinitely often will be committed infinitely often.
- *Strong fairness with respect to groups (SGF):* A group of processes that is ready to participate in ready interactions (consisting exactly of the processes in the group) infinitely often will be committed to some interaction (consisting exactly of the processes in the group) infinitely often.
- *Strong fairness with respect to interactions (SIF):* An interaction that is ready infinitely often will be committed infinitely often.

Note that *SGF* and *SIF* reduce to the same notion when for each set of processes P, there is at most one interaction consisting exactly of the processes in P.

Strong fairness is difficult to implement. In fact the only kind of strong fairness that has been shown to be implementable is *SPF* for the case of binary interactions [21, 23]. Tsay and Bagrodia [24] have shown that *SIF* is not achievable even for binary interactions. The impossibility result is proved by constructing an infinite history of three processes p, q, r that participate in three mutually conflicting binary interactions $\mathcal{I} = \{p, q\}$, $\mathcal{J} = \{q, r\}$, and $\mathcal{K} = \{r, p\}$. All the processes are initially idle and transit to the ready state together. However, on account of the asynchrony in the system, processes are only aware of the readiness of $\mathcal{I}$. The supposed algorithm is thus forced to commit $\mathcal{I}$ and processes p and q. After the commitment of $\mathcal{I}$ terminates, processes p and q become idle. Subsequently, process q becomes ready again. Since interaction $\mathcal{J}$ is the only ready interaction, it is committed. After the commitment of $\mathcal{J}$, processes q and r become idle. This brings us back to the initial system state at which all the processes are idle. This sequence of events is repeated forever. Thus $\mathcal{K}$, an infinitely often ready interaction, never commits and *SIF* is violated. If we consider the group $\{r, p\}$ then the same history also violates *SGF* and proves that *SGF* cannot be implemented. This proves the following theorem.

Theorem 3 SGF *and* SIF *are not implementable for the binary (and hence general) interaction problem.*

From this impossibility result, one can also show that *SPF* [2] is not implementable for general interactions. For this proof, we add one extra process r' to interaction $\mathcal{K}$. Process r' is continuously ready. We can then construct a history similar to the one constructed for the proof of Theorem 3 such that though process r' is continuously ready to participate in an infinitely ready interaction $\mathcal{K}$, it is never committed. Thus, *SPF* is violated. The following theorem follows.

Theorem 4 SPF *is not implementable for the general interaction problem.*

Given the above two theorems, it becomes an interesting question to find the strongest notion of fairness that is implementable. We define and implement such a fairness notion next.

3.1 A New Definition of Strong Fairness

The new notion of fairness is called *k-strong fairness (k-sf)* and is defined with respect to a subset of processes participating in an interaction. Let us say that a subset of processes is ready if each process in the subset is ready. Then the new definition of fairness is as follows.

Definition *A system satisfies k-sf iff some process in any continuously ready subset G of an infinitely often ready interaction $\mathcal{I}$, where $|P(\mathcal{I}\ ) - G| \leq k$, is eventually committed.*

In other words, let G be a subset of continuously ready processes such that an interaction $\mathcal{I}$, with $|P(I) - G| \leq k$, is ready infinitely often. Then we require that some process in G will be committed eventually to $\mathcal{I}$ or to some other interaction. For example, consider three interactions $\mathcal{X} = \{a, b\}$, $\mathcal{Y} = \{b, c\}$, and $\mathcal{Z} = \{a, c, d\}$ and a history in which process d remains ready forever and the following sequence of events repeats: processes a, b, and c become ready, $\mathcal{X}$ is committed, b becomes ready, and $\mathcal{Y}$ is committed. Then, this execution satisfies 1-*sf*. However, it does not satisfy 2-*sf* because $G = \{d\}$ is continuously ready, interaction $\mathcal{Z}$ is ready infinitely often, $|P(\mathcal{Z}) - G| \leq 2$, and process d is never committed to an interaction. Note that this fairness notion is similar to *SGF* in that we require fairness with respect to a set of processes. However, the definition of a group is different and is parameterized by k. The following properties can be proved for the new notion of fairness.

1. $(k + 1)$-*sf* implies k-*sf*.
2. 0-*sf* is equivalent to *WIF*.
3. k-*sf* is equivalent to *SPF* if $|P(\mathcal{I})| \leq k + 1$ for all $\mathcal{I}$.

The proof of the first two properties follows trivially from the definitions. The proof of the third property follows from letting G be a singleton set consisting of a process that is continuously ready. The first property implies that stronger fairness requirements are obtained by increasing k. The second property implies that *WIF* (i.e., 0-*sf*) is the weakest such fairness requirement and the third property implies that *SPF* is the strongest such fairness requirement. The third property also implies that for the case of binary interactions ($|P(\mathcal{I})| = 2|$), 1-*sf* is equivalent to *SPF*.

Theorem 5 *k-sf, $k \geq 2$, is not implementable.*

Proof: Consider the history constructed in Theorem 4 and the subset of processes $\{r'\}$. Though process r' is continuously ready and interaction $\mathcal{K}$ with $|P(\mathcal{K}\ ) - \{r'\}| = 2$ is infinitely often ready, process r' is never committed. This violates 2-sf and consequently, k-sf for any $k \geq 2$. □

Existing algorithms (including the algorithm presented in Section 2) show that 0-sf can be achieved for general interactions. We explore, 1-sf, the only remaining condition, in the remainder of this section. First, we prove that 1-sf is an adequate notion of fairness [1]. Next, we present algorithms achieving 1-sf for both binary and general interactions.

3.2 Adequacy of 1-sf

Apt, Francez, and Katz [1] have suggested a number of criteria appraising fairness notions under different models of computations. These criteria include *feasibility*, *equivalence robustness*, and *liveness enhancing*. Informally, a fairness notion F is *feasible* if for every program P, every finite initial segment of a legal computation of P can be extended to a legal computation that satisfies F. Fairness notion F is *equivalence robust* if for every program P and every two legal computations π and ρ that are equivalent under the model, π satisfies F iff ρ satisfies F. Fairness notion F is *liveness enhancing* if there is a program P such that some progress properties (such as program termination) can not be proved without F in the model of computation under consideration. A fairness notion is called *adequate* [1] if it satisfies all these three criteria.

It can be shown that 1-sf is adequate for the multi-party interaction problem under the asynchronous message passing system model. We only sketch the proofs of these properties. The ideas behind these proofs are essentially from [1]. To show that 1-sf satisfies feasibility, we construct an *explicit scheduler* that satisfies the following two requirements.

1. For any program P, any legal computation of P can be generated by the scheduler.
2. For any program P, any prefix of computation generated by the scheduler can be extended to a computation that satisfies 1-sf.

The implementation of the scheduler is as follows. It maintains properly initialized time-stamps for each interaction and proceeds in rounds. In each round, it decrements the time-stamp of each ready interaction. Then it selects the ready interaction, if any, with the minimal time-stamp to be committed (ties are resolved arbitrarily) and resets the corresponding time-stamp to an arbitrary value. Given any computation of program P, property 1 above can be satisfied by resetting the time-stamp of the just committed interaction to the number of rounds the interaction should remain ready without being selected for commitment after it becomes ready again. Therefore, any computation of P can be generated by the scheduler. Property 2 can be satisfied by resetting the time-stamp of the just committed interaction to a value larger than the current time-stamps of the interactions that need to be committed in order to satisfy 1-sf. Consequently, any interaction that needs to be committed in order to satisfy 1-sf will be selected by the scheduler since all other interactions will either be non-ready or have larger time-stamps.

For the proof of equivalence robustness, consider any computation π of an arbitrary program P that does not satisfy 1-sf. We show that any computation ρ equivalent to π does not satisfy 1-sf either. Since π does not satisfy 1-sf, there exists a set of continuously ready processes G and an infinitely often ready interaction $\mathcal{I}$ such that $|P(\mathcal{I}) - G| \leq 1$ and processes in G remain ready forever. Consequently, there is an infinite suffix π' of π such that processes in G remain ready at every state in π'. Since ρ is equivalent to π, there is an infinite suffix ρ' of ρ such that all processes in G remain ready at every state in ρ'. Since $|P(\mathcal{I}) - G| \leq 1$, either $P(\mathcal{I}) \subseteq G$ or there exists exactly one process p in the set $P(\mathcal{I}) - G$. In the former case, interaction $\mathcal{I}$ is ready continuously and hence infinitely often in ρ'. In the latter case, process p is ready infinitely often in π' and hence infinitely often in ρ'. Since the remaining processes of interaction $\mathcal{I}$ are continuously ready in ρ', $\mathcal{I}$ must again be ready infinitely often in ρ'. Therefore, though processes in G are continuously ready and $\mathcal{I}$ is ready infinitely often in ρ, processes in G are not committed in the suffix ρ'. Consequently, the computation ρ does not satisfy 1-sf.

For the proof of liveness enhancing, we use the following example consisting of three processes p, q, r and three interactions $\mathcal{I} = \{p, q\}$, $\mathcal{J} = \{q, r\}$, and $\mathcal{K} = \{p, r\}$. The code of the processes is shown in Figure 4.

```
Process p: repeat
               transit to ready;
               commit to interaction I or K
           until interaction K is committed

Process q: repeat
               transit to ready;
               commit to interaction I or J
           until interaction J is committed

Process r: transit to ready;
           commit to interaction J or K;
           transit to ready;
           commit to interaction J or K
```

Figure 4: An example for liveness enhancing

The requirement of 1-sf (i.e., *SPF* as all the interactions are binary) is needed in Figure 4 in order to ensure that process r will eventually commit to some interaction. Since the processes terminate only if process r commits, it follows that the assumption of 1-sf is needed to ensure that the processes terminate. Thus, 1-sf is liveness enhancing and we have the following theorem.

Theorem 6 *Fairness notion* 1-sf *is adequate for the interaction problem under the asyn-*

chronous message passing system model.

3.3 1-sf for Binary Interaction

idle to *ready* transition	$nx := next(Q)$; **send** $\langle direct\text{-}req\rangle$ to nx
Receive $\langle no\rangle$ from nx	$nx := null$; **send** $\langle ready\rangle$ to all associated coordinators
Receive $\langle yes\rangle$ from nx	commit to interaction $\{p, nx\}$; $nx := null$
Receive $\langle request\rangle$ or $\langle direct\text{-}req\rangle$ when $nx \neq null$	**send** $\langle no\rangle$ to sender
Receive $\langle direct\text{-}req\rangle$ from q when $nx = null$	**if** $ready \ \wedge \ \neg flag$ **then** **send** $\langle yes\rangle$ to q; commit to interaction $\{p, q\}$ **elsif** $ready$ **then** $pending := pending \cup \{q\}$ **else send** $\langle no\rangle$ to q **fi**
Receive $\langle request\rangle$ from c_I when $nx = null$	**if** $ready$ **then** **send** $\langle yes\rangle$ to c_I; $flag :=$ true **else send** $\langle no\rangle$ to c_I **fi**
Receive $\langle commit\rangle$ from c_I when $nx = null$	**send** $\langle ack\rangle$ to c_I; **send** $\langle no\rangle$ to all processes in $pending$; $flag :=$ false; $pending := \{\}$; commit to interaction $\mathcal{I}$
Receive $\langle no\text{-}commit\rangle$ from c_I when $nx = null$	$flag :=$ false; **if** $pending \neq \{\}$ **then** select any $r \in pending$; **send** $\langle yes\rangle$ to r and $\langle no\rangle$ to all others; $pending := \{\}$; commit to interaction $\{p, r\}$ **fi**

Figure 5: Algorithm 2 for process p

Here we present an algorithm that implements 1-sf for binary interactions. Since 1-sf is equivalent to *SPF* for binary interactions, we will be using the term *SPF* in this subsection. The algorithm is composed of two phases. In the first phase, a process attempts commitment directly with neighboring processes on a round-robin basis. In the second phase, processes attempt commitment with the help of coordinators and philosophers. As in Algorithm 1, the use of coordinators and philosophers ensures that *WIF* is satisfied. When the round-robin mechanism of the first phase is added, the algorithm also satisfies SPF. The resulting algorithm, Algorithm 2, is shown in figures 5 and 6.

In order to implement the first phase, each process p maintains a circular queue Q of interactions in which it participates. A local variable nx points to the process with which the process is currently attempting commitment. In the beginning of the first phase, this variable is set to q where $\{p, q\}$ is the next interaction in queue Q. In order to commit with process q, process p sends a $\langle direct\text{-}req \rangle$ message to process q. Process q responds to this message either by sending a $\langle yes \rangle$ or a $\langle no \rangle$ message. The interaction $\{p, q\}$ is committed if a $\langle yes \rangle$ message is received. Otherwise, the second phase is begun by sending a $\langle ready \rangle$ message to the coordinators of the interactions in which process p participates. In either case, variable nx is set to a special value $null$. If process p receives a $\langle direct\text{-}req \rangle$ message in the first phase, then it responds with a $\langle no \rangle$ message.

Receive $\langle ready \rangle$ from p	**insert** p into $ReadySet$
Receive $\langle ready \rangle$ from q	**insert** q into $ReadySet$

When $ReadySet = \{p, q\}$	set f_I to hungry;
	wait until f_I is eating;
	send $\langle request \rangle$ to p and q;
	wait until each process replies;
	if both replies are $\langle yes \rangle$ **then**
	    send $\langle commit \rangle$ to p and q;
	    wait for $\langle ack \rangle$ from p and q;
	    $ReadySet := \{\}$
	elsif one process, say p, replies $\langle yes \rangle$ **then**
	    send $\langle no\text{-}commit \rangle$ to p;
	    $ReadySet := \{p\}$
	else $ReadySet := \{\}$ **fi**;
	set f_I to thinking

Figure 6: Algorithm 2 for c_I where $P(\mathcal{I}) = \{p, q\}$

For the second phase, a coordinator c_I and a philosopher f_I are selected for each interaction $\mathcal{I}$. As in Algorithm 1, philosophers execute an underlying dining philosophers algorithm and conflict with each other if the corresponding interactions conflict. Consider an interaction $\mathcal{I} = \{p, q\}$. Coordinator c_I inserts $\langle ready \rangle$ requests from processes p and q into a set $ReadySet$. Once both processes have sent $\langle ready \rangle$ requests, the coordinator sets philosopher f_I to hungry. When philosopher f_I transits to eating, the coordinator sends $\langle request \rangle$ messages to processes p and q. On receiving a $\langle request \rangle$, a process always replies with a $\langle no \rangle$ message if it is in the first phase. If a process is in the second phase, then it replies with a $\langle yes \rangle$ message if it is $ready$. Otherwise, it replies with a $\langle no \rangle$ message. If the coordinator receives $\langle yes \rangle$ messages from both the processes p and q then it sends a $\langle commit \rangle$ message to them, waits for acknowledgements to be returned and sets $ReadySet$ to $\{\}$. If the coordinator receives a $\langle yes \rangle$ message from only one of the process, say p, then it responds with a $\langle$*no-commit*$\rangle$ message and sets $ReadySet$ to $\{p\}$. If the coordinator receives $\langle$*no*$\rangle$ messages from both processes then it sets $ReadySet$ to $\{\}$. In all the three cases, philosopher f_I is set to the thinking state.

It is possible for a process p in the second phase to receive a $\langle$*direct-req*$\rangle$ message from a process q in the first phase. In that case if process p is ready and has not responded with a $\langle$*yes*$\rangle$ message to some coordinator (this is signaled by $flag$ being true) then it responds with a $\langle$*yes*$\rangle$ message and commits to interaction $\{p, q\}$. Otherwise, if process p is ready and has responded with a $\langle$*yes*$\rangle$ message to some other coordinator then q is saved in a set *pending* for future consideration. If process p is not ready then it responds with a $\langle$*no*$\rangle$ message.

On receiving a $\langle$*commit*$\rangle$ or a $\langle$*no-commit*$\rangle$ message, process p completes sending replies to pending $\langle$*direct-req*$\rangle$ messages. If a $\langle$*commit*$\rangle$ message is received then the process sends an $\langle$*ack*$\rangle$ message to the coordinator and a $\langle$*no*$\rangle$ message to all the processes in set *pending*. If a $\langle$*no-commit*$\rangle$ message is received then process p selects some process r in *pending*, and sends a $\langle$*yes*$\rangle$ to it and $\langle$*no*$\rangle$ to the other processes in *pending*, and commits to interaction $\{p, r\}$. In both the cases, $flag$ is set to false and *pending* is set to $\{\}$.

3.4 Correctness and Performance Evaluation

The following lemma can be proved in a manner analogous to Lemma 1.

Lemma 2 *The eating period of a philosopher is finite and at most* 4ν.

The following lemma bounds the duration of the first phase at a process.

Lemma 3 *The duration of the first phase (i.e., the duration for which variable nx remains non-null) is at most* 5ν *time units.*

Proof: According to the algorithm, when a process p enters the first phase, it sends a $\langle$*direct-req*$\rangle$ message to some process q such that the interaction $\{p, q\}$ is at the head of the queue Q. If q is in the first phase when it receives the $\langle$*direct-req*$\rangle$ message then it will return a $\langle$*no*$\rangle$ message to p and the first phase ends within 2ν time units. On the other hand, if q is not in the first phase when it receives the $\langle$*direct-req*$\rangle$ message, then it will either insert p in the set *pending*, or respond with a $\langle$*yes*$\rangle$ or a $\langle$*no*$\rangle$ message. In the latter case, the first phase ends when p receives the response within 2ν time units. Now, consider the former case in which p is inserted in *pending*. In this case, $flag$ must be true at process q, i.e, process q must have responded with a $\langle$*yes*$\rangle$ message to some coordinator c_J. This $\langle$*yes*$\rangle$ message arrives at c_J in at most ν time units and all other $\langle$*yes*$\rangle$ or $\langle$*no*$\rangle$ messages arrive at c_J in at most 2ν time units. Then c_J either sends a $\langle$*commit*$\rangle$ or a $\langle$*no-commit*$\rangle$ message to q in another ν time units. If c_J sends a $\langle$*commit*$\rangle$ message to q then q commits and sends a $\langle$*no*$\rangle$ message to all the processes in *pending*, and in particular to p. If c_J sends a $\langle$*no-commit*$\rangle$ message to q then q either sends a $\langle$*yes*$\rangle$ or a $\langle$*no*$\rangle$ message to p. Adding the above time units, p receives a $\langle$*yes*$\rangle$ or a $\langle$*no*$\rangle$ message within 5ν time units and ends its first phase. $\square$

Theorem 7 *Algorithm 2 satisfies the requirements of mutual exclusion, synchronization, and* SPF*.*

Proof: To prove mutual exclusion, it is sufficient to show that no process commits to more than one interaction at the same time. Observe that a process p may commit to an interaction either in its first phase or in its second phase. While process p is in its first phase, it can only commit to the interaction it chooses from the head of the queue Q. Therefore, it can commit to at most one interaction. In the second phase, process p can commit to an interaction either by sending a $\langle yes \rangle$ message to a process q in response to a $\langle direct\text{-}req \rangle$ message or by sending a $\langle yes \rangle$ message to a coordinator c_I in response to a $\langle request \rangle$ message. In the former case, process p has no outstanding $\langle yes \rangle$ messages to coordinators on account of $flag$ being false. Therefore, it has not committed to some other interaction earlier. Moreover, it will not respond with a $\langle yes \rangle$ message to any other process or interaction as its state is changed to *commit*. In the latter case, process p does not have any outstanding $\langle yes \rangle$ messages to coordinators since coordinator c_I sends $\langle request \rangle$ messages only while philosopher f_I is eating and from the correctness of the underlying dining philosophers algorithm, conflicting philosophers do not eat at the same time. Since philosopher f_I remains eating until an $\langle ack \rangle$ message is returned by process p, process p does not receive any $\langle request \rangle$ messages from coordinators until a $\langle commit \rangle$ message is received from coordinator c_I. Moreover, since $flag$ is set to true at process p, it will not respond with a $\langle yes \rangle$ message to any other process that wants to commit in the first phase. The mutual exclusion property follows.

To prove synchronization, we must show that interaction $\{p, q\}$ is not committed unless both p and q are ready. Note that interaction $\{p, q\}$ can be committed either when one of the processes is in its first phase or when both the processes are in the second phase. In the former case, one process must have sent a $\langle direct\text{-}req \rangle$ message to the other and received a $\langle yes \rangle$ response. This implies that both the processes must be ready. In the latter case, both processes must have received a $\langle request \rangle$ message from the coordinator and responded with a $\langle yes \rangle$ message. From the algorithm, both the processes must be ready when they responded with $\langle yes \rangle$ messages. As shown earlier, coordinators of conflicting interactions send $\langle request \rangle$ messages in a mutual exclusive manner. Therefore, a process cannot change its state while a $\langle yes \rangle$ message is outstanding. Therefore, the processes are also in a ready state when they receive a $\langle commit \rangle$ message from the coordinator. The proof follows.

Finally, we show that the requirement of *SPF* is satisfied. For the sake of contradiction, assume that process p is ready to participate in ready interactions infinitely often but is committed only a finite number of times. This implies that there exists a state after which process p is ready continuously and some interaction $\{p, q\}$ is ready infinitely often. If process q is also continuously ready then by Lemma 3 eventually both the processes will enter the second phase and send $\langle ready \rangle$ messages to the coordinator. Upon receiving these messages, the coordinator will set the corresponding philosopher to hungry. By the correctness of the underlying dining philosophers algorithm, the philosopher will eventually eat. Then the coordinator will send $\langle request \rangle$ messages to processes p and q. From the algorithm, processes p and q will respond with a $\langle yes \rangle$ message and contrary to our assumption, the interaction $\{p, q\}$ will be committed.

Now, consider the other case in which process q is not ready continuously. Since the interaction $\{p, q\}$ is ready infinitely often, process q must transit to the ready state from the idle state infinitely often. This implies that process q executes its first phase infinitely often. By the management of queue Q, process q will set $nx := p$ infinitely often and send a $\langle$*direct-req*$\rangle$ message to process p infinitely often. By Lemma 3, at least one of these messages is received at process p when it is no longer in its first phase. If $flag$ is false at process p when such a message is received then contrary to our assumption, interaction $\{p, q\}$ is committed. So, consider the other case in which $flag$ is true at process p when such a message is received. In that case process p must have sent a $\langle$*yes*$\rangle$ message to some coordinator c_I and consequently, q is inserted into set *pending*. Since each eating period is finite, process p will eventually receive a $\langle$*commit*$\rangle$ or a $\langle$*no-commit*$\rangle$ message from c_I. In the former case, process p will be committed, contradicting our assumption that process p is continuously ready. In the latter case, process p will choose some process r from *pending* (which is non-empty as q belongs to it) and commit to interaction $\{p, r\}$. This again contradicts our assumption that process p is continuously ready. The proof follows. $\Box$

Theorem 8 *For any f and g, if the response time of the underlying dining philosophers algorithm is $f(\delta)*\tau+g(\delta)*\nu$, then the response time of Algorithm 2 is $4f(\delta)*\nu+g(\delta)*\nu+9\nu$. Also, if the message complexity of the underlying dining philosophers algorithm is $M(\delta)$ then the message complexity of Algorithm 2 is $M(\delta)+14$.*

Proof: The proof of the response time bound is similar to that in Theorem 2 except that an extra 5ν time units are added on account of the first phase. The proof of the message complexity bound follows as at most 4 messages are sent in the first phase and at most 10 messages are sent in the second phase per eating period. $\Box$

If we use the algorithm presented by us in [7] as the underlying dining philosophers algorithm ($f(\delta) = g(\delta) = O(\delta^2)$ and $M(\delta) = O(\delta)$) then the response time of the composite algorithm is $O(k^2\nu)$ and the message complexity of the composite algorithm is $O(k)$. In comparison, the response time of the algorithms by Sistla [21] and Tsay and Bagrodia [23] is $O(k^3\nu)$. The message complexity of all the three algorithms is $O(k)$. It should be noted that the response time bound of Algorithm 2 is of the same order of magnitude as the best known algorithms [4, 23] satisfying the weaker requirement of *WIF*.

3.5 1-sf for General Interactions

The algorithm for the case of general interactions is once more composed of two phases: a first phase in which processes attempt commitment directly with each other and a second phase in which processes attempt commitment with the help of coordinators and philosophers. As before, a coordinator c_I and a philosopher f_I are defined for each interaction $\mathcal{I}$. In addition, for each interaction $\mathcal{I}$ and for each process $p \in \mathcal{I}$, we define a superprocess p_I and a philosopher f_{p_I} representing the set of processes $\mathcal{I} - \{p\}$. The job of this superprocess and the corresponding philosopher is to check the readiness of the processes in the set $\mathcal{I} - \{p\}$ without any interference. As before, two philosophers conflict with each other if the sets of processes they represent share a common process. As an example of the transformation scheme, consider the 3 interactions $\mathcal{X}, \mathcal{Y}$, and $\mathcal{Z}$ shown in Figure 7. Besides coordinators c_X, c_Y, c_Z and philosophers f_X, f_Y, f_Z, seven superprocesses and seven new philosophers are also defined. For example, for interaction $\mathcal{X}$, we define two superprocesses a_X, b_X and two philosophers f_{a_X}, f_{b_X}. Superprocess a_X and philosopher f_{a_X} represent the set of processes $P(\mathcal{X}) - \{a\} = \{b\}$ whereas superprocess b_X and philosopher f_{b_X} represent the set of processes $P(\mathcal{X}) - \{b\} = \{a\}$. The other superprocesses and philosophers are defined similarly.

Multi-party interactions:

$\mathcal{X} = \{a, b\}, \mathcal{Y} = \{b, c\}, \mathcal{Z} = \{a, c, d\}$

Coordinators:

$c_X,\ c_Y,\ c_Z$

Philosophers for interactions:

$f_X,\ f_Y,\ f_Z$

Superprocesses:

$a_X, b_X, b_Y, c_Y, a_Z, c_Z, d_Z$

Philosophers for superprocesses:

$f_{a_X},\ f_{b_X},\ f_{b_Y},\ f_{c_Y},\ f_{a_Z},\ f_{c_Z},\ f_{d_Z}$

Figure 7: An example of a transformation

The new algorithm, Algorithm 3, for general interaction is shown in figures 8, 9, and 10. As before, each process p maintains a circular queue Q of interactions in which it participates. If the interaction at the head of the queue is $\mathcal{I}$ then the function *next* returns the identity of the superprocess p_I. Upon becoming ready, process p first attempts commitment with interaction $\mathcal{I}$ by sending a $\langle$*direct-req*$\rangle$ message to superprocess p_I. When superprocess p_I receives this message, it checks the state of all the processes in the set $P(\mathcal{I}) - \{p\}$. In order to avoid any interference, this checking is carried out by setting philosopher f_{p_I} to hungry, and waiting until it starts eating. A $\langle$*commit*$\rangle$ message is sent to process p provided all the processes in $P(\mathcal{I}) - \{p\}$ are ready; a $\langle$*no-commit*$\rangle$ message is sent otherwise. In case $\langle$*commit*$\rangle$ messages are sent, the superprocess waits until acknowledgements are returned. In any case, the state of the philosopher f_{p_I} is set to thinking. If process p fails to commit in the first phase (i.e., a $\langle$*no-commit*$\rangle$ message is received from superprocess p_I), it sets nx to *null*, sends a $\langle$*ready*$\rangle$ message to all coordinators, and transits to the second phase. Symbol d in Figure 8 denotes either a superprocess or a coordinator.

The algorithm for a coordinator c_I is as in Algorithm 2. Once it receives a $\langle$*ready*$\rangle$ message from all the processes in $P(\mathcal{I})$, it sets philosopher f_I to hungry and waits until the philosopher starts eating. Then it sends a $\langle$*request*$\rangle$ message to every process to check if they are ready. If it receives $\langle yes \rangle$ messages from all the processes, it commits interaction $\mathcal{I}$. Otherwise, it sets $ReadySet$ to the set of processes that replied with $\langle yes \rangle$ messages. The state of philosopher f_I is then set to thinking.

$idle$ to $ready$ transition	$nx := next(Q)$; **send** $\langle direct\text{-}req \rangle$ to nx
Receive $\langle no\text{-}commit \rangle$ from superprocess p_J when $nx \neq null$	$nx := null$; **send** $\langle ready \rangle$ to all associated coordinators
Receive $\langle commit \rangle$ from superprocess p_J when $nx \neq null$	**send** $\langle ack \rangle$ to p_J; commit to interaction $\mathcal{J}$; $nx := null$
Receive $\langle request \rangle$ from d when $nx \neq null$	**send** $\langle no \rangle$ to d
Receive $\langle request \rangle$ from d when $nx = null$	**if** $ready$ **then send** $\langle yes \rangle$ to d; **else send** $\langle no \rangle$ to d **fi**
Receive $\langle commit \rangle$ from d when $nx = null$	**send** $\langle ack \rangle$ to d; commit to interaction represented by d

Figure 8: Algorithm 3 for process p

Receive $\langle direct\text{-}req \rangle$ from p	**set** f_{p_I} to hungry;
	wait until f_{p_I} is eating;
	send $\langle request \rangle$ to all processes in $P(\mathcal{I}) - \{p\}$;
	wait until each process replies;
	if all the replies are $\langle yes \rangle$ **then**
	    send $\langle commit \rangle$ to every process in $P(\mathcal{I})$;
	    wait for $\langle ack \rangle$ from every process in $P(\mathcal{I})$
	else send $\langle no\text{-}commit \rangle$ to process p **fi**;
	set f_{p_I} to thinking

Figure 9: Algorithm for 3 superprocess p_I

3.6 Proof of Correctness

Since the underlying dining philosophers algorithm requires that eating periods be finite, we first prove the following lemma.

Lemma 4 *The eating period of any philosopher in Algorithm 3 is finite and at most* 4ν.

Proof: For both types of philosophers (i.e., f_{p_I} or f_I), the eating period consists of sending $\langle request \rangle$ messages to processes, waiting for replies, and possibly sending $\langle commit \rangle$ messages and waiting for acknowledgements. Since processes respond immediately on receiving $\langle request \rangle$ or $\langle commit \rangle$ messages, the duration of any eating period is at most 4ν. $\square$

Receive $\langle ready \rangle$ from process p	**insert** p into $ReadySet$

When $ReadySet$ = P($\mathcal{I}$)	**set** f_I to hungry;
	wait until f_I is eating;
	send $\langle request \rangle$ to every $q \in P(\mathcal{I})$;
	wait until each process replies;
	if all replies are $\langle yes \rangle$ **then**
	    $ReadySet := \{\}$;
	    send $\langle commit \rangle$ to every $q \in P(\mathcal{I})$;
	    wait for $\langle ack \rangle$ from every process in $P(\mathcal{I})$
	else $ReadySet$:= the set of processes
	        that reply with $\langle yes \rangle$ messages **fi**;
	set f_I to thinking

Figure 10: Algorithm 3 for coordinator c_I

Unlike Algorithm 2, the duration of the first phase is no longer $O(\nu)$ since it involves the hungry to eating transition of a superprocess. However, it is still finite.

Lemma 5 *The duration of the first phase in Algorithm 3 is finite.*

Proof: The first phase begins when a process p becomes ready and sends a $\langle direct\text{-}req \rangle$ message to a superprocess p_I and it ends when the superprocess replies with either a $\langle commit \rangle$ or a $\langle no\text{-}commit \rangle$ message. According to the algorithm, when the superprocess receives the $\langle direct\text{-}req \rangle$ message, it sets the state of the corresponding philosopher to hungry and waits until the philosopher starts eating. From the correctness of the underlying dining philosophers algorithm, the philosopher will eat eventually. After that the superprocess sends $\langle request \rangle$ messages to processes in $P(\mathcal{I}) - \{p\}$. These processes respond immediately with $\langle yes \rangle$ or $\langle no \rangle$ messages. Based on these responses, the superprocess sends a $\langle commit \rangle$ or a $\langle no\text{-}commit \rangle$ message to process p. This ends the first phase at process p. $\square$

Let δ be the degree of the conflict graph of the philosophers (of both types f_I and f_{p_I}) produced by the transformation in Algorithm 3.

Corollary 1 *For any f and g, if the response time of the underlying dining philosophers algorithm is $f(\delta) * \tau + g(\delta) * \nu$ then the duration of the first phase is at most $4f(\delta) * \nu + g(\delta) * \nu + 4\nu$.*

The following theorem establishes the correctness of Algorithm 3.

Theorem 9 *Algorithm 3 satisfies the requirements of mutual exclusion, synchronization, and 1-sf.*

Proof: Consider the mutual exclusion requirement first. It is sufficient to show that no process commits to more than one interaction at the same time. Observe that a process p may commit to an interaction either in its first phase or in its second phase. While process

p is in its first phase, it can only commit to the interaction it chooses from the head of the queue Q. Therefore, it can commit to at most one interaction. In the second phase, process p can commit to an interaction either by sending a $\langle yes \rangle$ message to a superprocess q_I (or a coordinator c_I) in response to a $\langle request \rangle$ message. In both the cases, process p does not have any outstanding $\langle yes \rangle$ messages to coordinators or superprocesses since superprocess q_I (or coordinator c_I) sends $\langle request \rangle$ messages only while philosopher f_{q_I} (or f_I) is eating and from the correctness of the underlying dining philosophers algorithm, conflicting philosophers do not eat at the same time. After process p receives a $\langle commit \rangle$ message, philosopher f_{q_I} (or f_I) remains eating until an $\langle ack \rangle$ message is returned by process p. Consequently, process p does not receive any $\langle request \rangle$ messages until it commits and once it commits, it responds with a $\langle no \rangle$ message to all such $\langle request \rangle$ messages. The mutual exclusion property follows.

Next, we consider the synchronization requirement. We have to show that an interaction $\mathcal{I}$ is committed only if all the processes in $P(\mathcal{I})$ are ready. According to the algorithm, interaction $\mathcal{I}$ is committed either by a superprocess p_I or by the coordinator c_I. In the former case, philosopher f_{p_I} must be in the eating state and superprocess p_I must have determined that all the processes in $P(\mathcal{I}) - \{p\}$ are ready. In the latter case, philosopher f_I must be in the eating state and coordinator c_I must have determined that all the processes in $P(\mathcal{I})$ are ready. So, the only way the synchronization requirement can be violated is by a ready process becoming non-ready while the coordinator or the superprocess is checking the states of the other processes. Now, a ready process can become non-ready only if some neighboring interaction is committed in the meantime. (Note that by the algorithm, a process p that has sent a $\langle direct\text{-}req \rangle$ message to a superprocess p_I remains ready until it receives a response from the superprocess.) But this cannot happen because an interaction is committed only while the corresponding philosopher is eating and because from the mutual exclusion property of the underlying solution neighboring philosophers do not eat at the same time. The proof follows.

Finally, we consider the requirement of 1-sf. For the sake of contradiction, assume that G is a set of continuously ready processes, there is an infinitely often ready interaction $\mathcal{I}$ such that $|P(\mathcal{I}) - G| \leq 1$, and processes in G are never committed. Since, $|P(\mathcal{I}) - G| \leq 1$, either $P(\mathcal{I}) \subseteq G$ or $|P(\mathcal{I}) - G| = 1$. In the former case, all the processes in $\mathcal{I}$ are continuously ready. Since by Lemma 5, each process in $\mathcal{I}$ will eventually be in the second phase, the coordinator c_I will eventually receive $\langle ready \rangle$ messages from the processes, and commit the interaction $\mathcal{I}$ and some processes in G. This contradicts our assumption that processes in G are never committed. Now, consider the other case in which $|P(\mathcal{I}) - G| = 1$. Let q be the process in the set $P(\mathcal{I}) - G$ that is ready infinitely often. If process q is continuously ready then reasoning as in the previous case will establish that the coordinator c_I will eventually commit the interaction $\mathcal{I}$ and some processes in G, contradicting our assumption. Therefore, assume that process q transits to the ready state infinitely often. By the management of the queue Q, q will set $nx := q_I$ and send a $\langle direct\text{-}req \rangle$ message to superprocess q_I infinitely often. Each time superprocess q_I will set philosopher f_{q_I} to hungry, wait until the philosopher is eating (this is ensured by the underlying dining philosophers algorithm), and send $\langle request \rangle$ messages to processes in $P(\mathcal{I}) - \{q\}$. By Lemma 5, eventually all these messages will be received by processes in $P(\mathcal{I}) - \{q\}$ after they are no longer in the first phase. Therefore, eventually all these processes will respond with $\langle yes \rangle$ messages and consequently superprocess q_I will commit interaction $\mathcal{I}$. This commits some processes in G, contradicting our assumption that processes in G are never committed. The proof follows. $\Box$

Theorem 10 *For any f and g, if the response time of the underlying dining philosophers algorithm is $f(\delta)*\tau+g(\delta)*\nu$, then the response time of Algorithm 3 is $8f(\delta)*\nu+2g(\delta)*\nu+8\nu$. Also, if the message complexity of the underlying dining philosophers algorithm is $M(\delta)$ then the message complexity of Algorithm 3 is $(m+1)*M(\delta)+2m^2+5m$.*

Proof: The proof of the response time bound is similar to that in Theorem 2 except that an extra $4f(\delta)*\nu+g(\delta)*\nu+4\nu$ time units are added on account of the first phase.

The total number of messages transmitted on account of a process in the first phase is $M(\delta)+4m$ if the interaction at the head of the queue is committed and $M(\delta)+2m$ if the interaction at the head of the queue is not committed. In the latter case, a process proceeds onto the second phase where the total number of messages transmitted per eating period of a coordinator is bounded by $M(\delta)+5m$. Thus, if an interaction is committed during the first phase of a participating process, then the message complexity is given by $(m-1)*(M(\delta)+2m)+M(\delta)+4m$. Otherwise, the message complexity is given by $m*(M(\delta)+2m)+M(\delta)+5m$. $\Box$

Next, we compute δ, the degree of the conflict graph generated by transformation in Algorithm 3. Assume as before that each process participates in at most k interactions and an interaction consists of at most m processes. Since at most m superprocesses are generated for each interaction and since a process participating in the interaction belongs to all but one of these superprocesses, a process belongs to at most $m-1$ superprocesses for each interaction in which it participates. Therefore, a process belongs to at most $k(m-1)$ superprocesses. Consider any process $p \in P(\mathcal{I})$. The number of edges for philosopher f_I on account of process p is at most $k-1$ on account of conflicting coordinators and at most $k(m-1)$ on account of conflicting superprocesses. Thus, the total number of edges for philosopher f_I on account of each process in $P(\mathcal{I})$ is $k(m-1)+k-1 = km-1$. Since philosopher f_I represents at most m processes, it has at most $m*(km-1)$ edges. A similar argument shows that a philosopher f_{p_I} has at most $(m-1)*k$ edges on account of conflicting coordinators and at most $(m-1)*k(m-1)$ edges on account of conflicting superprocesses. Therefore, a philosopher f_{p_I} has at most $(m-1)*km$ edges. Consequently, the degree of the conflict graph is at most $m*(km-1)$. If we use the algorithm presented by us in [7] as the underlying dining philosophers algorithm ($f(\delta)=g(\delta)=O(\delta^2)$ and $M(\delta)=O(\delta)$) then the response time of the composite algorithm is $O(m^4k^2\nu)$ and the message complexity of the composite algorithm is $O(m^3k)$.

4 Discussion

In this paper we considered the implementation of synchronous communication primitives over asynchronous message-passing networks. In the first half of the paper, we presented an efficient solution to the zeroth-order interaction problem (committee coordination problem) and the first-order interaction problem. The response time of the algorithms was an improvement over the the existing solutions. In the second half of the paper, we considered the implementation of strong fairness for the zeroth-order version of the problem. We defined a new notion of fairness called k-sf. We proved that this kind of fairness cannot be implemented if $k \geq 2$. Then, we considered 1-sf. We showed that for the case of binary interactions, 1-sf is equivalent to *SPF*, the only kind of fairness shown to be implementable. Then, we showed that 1-sf is feasible, equivalence robust, and liveness enhancing and consequently, an *adequate* notion of fairness [1]. Finally, we presented algorithms that implement 1-sf for both binary and general interactions. Both the algorithms were constructed by combining an underlying solution to the dining philosophers problem with a queuing mechanism at each process. The former ensured *WIF* and the latter ensured 1-sf. The resulting algorithm for binary interactions had a better response time than existing solutions and the resulting algorithm for general interactions was the first one to satisfy a fairness condition stronger than *WIF*.

References

[1] K. R. Apt, N. Francez, and S. Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2:226–241, 1988.

[2] P. C. Attie, N. Francez, and O. Grumberg. Fairness and hyperfairness in multi-party interactions. In *Proceedings of the 8th Annual ACM Symposium Principles of Programming Languages*, pages 292–305, 1990.

[3] R. J. R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, 1988.

[4] R. L. Bagrodia and Y. Tsay. An efficient algorithm for fair interprocess synchronization. Unpublished Manuscript. 1989.

[5] G. N. Buckley and A. Silberschatz. An effective implementation for the generalized input-output construct of CSP. *ACM Transactions on Programming Languages and Systems*, 5(2):223–235, 1982.

[6] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison Wesley Pub. Co., Reading, Mass., 1988.

[7] M. Choy and A. K. Singh. Efficient fault tolerant algorithms for resource allocation in distributed systems. In *Proceedings of the 21th Annual ACM Symposium on Theory of Computing*, 1992.

[8] M. Choy and A. K. Singh. Tight lower bounds on failure locality of distributed synchronization. In *Proceedings of the Thirtieth Annual Allerton Conference on Communication, Control, and Computing*, 1992.

[9] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[10] I. R. Forman. On the design of large distributed systems. In *Proceedings of the First International Conference on Computer Languages*, pages 84–95, Miami, Florida, 1986.

[11] N. Francez. *Fairness*. Springer-Verlag, 1986.

[12] N. Francez and I. R. Forman. *Interacting Processes: A Multiparty Approach to Coordinated Distributed Programming.* Forthcoming book, 1990.

[13] N. Francez, B. Hailpern, and G. G. Taubenfeld. Script: A communication abstraction mechanism. *Science of Computer Programming*, 6(1):35–88, 1986.

[14] Y. J. Joung and S.A. Smolka. Coordinating first-order multiparty interactions. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 209–220, 1991.

[15] D. Kumar. An algorithm for N-party synchronization using tokens. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 320–327, 1990.

[16] N. Lynch. Fast allocation of nearby resources in a distributed system. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, pages 70–81, 1980.

[17] United States Department of Defense. *Reference manual for the Ada programming language.* Springer-Verlag, New York, 1983.

[18] M. H. Park and M. Kim. A distributed synchronization scheme for fair multi-process handshakes. *Information Processing Letters*, (24):131–138, 1990.

[19] S. Ramesh. A new and efficient implementation of multiprocess synchronization. In *PARLE: Parallel Architectures and Languages Europe. Lecture Notes in Computer Sciences 259*, pages 387–401, Springer-Verlag, Berlin, 1987.

[20] S. Ramesh and S. L. Mehndiratta. A methodology for developing distributed programs. *IEEE Transactions on Software Engineering*, 13(8):967–976, 1987.

[21] A. P. Sistla. Distributed algorithms for ensuring fair interprocess communications. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, pages 266–277, 1984.

[22] E. Styer and G. Peterson. Improved algorithms for distributed resource allocation. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pages 105–116, 1988.

[23] Y. Tsay and R. L. Bagrodia. A real-time algorithm for fair interprocess synchronization. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, 1992.

[24] Y. Tsay and R. L. Bagrodia. Some impossibility results in interprocess synchronization. *Technical Report CSD-890059, Computer Science Department, UCLA*, 1989.