Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

AN EFFICIENT ALGORITHM FOR FAIR INTERPROCESS SYNCHRONIZATION

Rajive L. Bagrodia
Yih-Kuen Tsay

August 1990
CSD-900021

An Efficient Algorithm for Fair Interprocess Synchronization[1]

Rajive L. Bagrodia
Yih-Kuen Tsay

3531 Boelter Hall
Computer Science Department
University of California at Los Angeles
Los Angeles, CA 90024.
Tel: (213) 825-0956

bagrodia@cs.ucla.edu
yihkuen@cs.ucla.edu

Abstract. An efficient algorithm for binary interactions with strong process fairness is described. Unlike previous algorithms for this problem, the complexity of our algorithm is independent of the total number of processes in the system and dependent only on the number of communication partners for every process. The design process of the algorithm is elaborated using reasoning about knowledge; necessary and sufficient conditions that must be satisfied by any algorithm that implements fair synchronization are derived in the form of knowledge predicates.

[1]This research was supported by a grant from NSF (CCR 88 10376)

1 Introduction

The multiway interaction problem abstracts the basic issues, namely synchronization and mutual exclusion, in implementing the symmetric, nondeterministic synchronous communication constructs of programming languages like CSP [Hoa78], Script [FHT86], and IP [Fra89] on a distributed architecture. An anthropomorphic version of this problem, called committee coordination, can be found in [CM88].

Consider a set of processes and a set of interactions defined among them. Each interaction is a nonempty subset of processes representing some synchronization activity of its members. A process can be in *active*, *idle*, or *commit* state. An active process may autonomously become idle and wait to participate in some interaction. (Note that, in general, it is impossible to determine when, or if, an active process will become idle.) An interaction is *enabled* if all of its members are idle; it is *disabled* otherwise. Only enabled interactions can be started — synchronization. An idle process will transit to commit state only after an interaction of which it is a member is started. A process can participate in at most one interaction — mutual exclusion. A process in commit state can become active only after the interaction in which it participates is terminated — another synchronization.

The binary interaction problem is a special case where each interaction has exactly two members.

The interaction problems are significantly different from the standard resource allocation problems [Lyn80]. In the latter, a process requests a resource and blocks until the resource is available; however, in the interaction problem if an interaction is viewed as a resource, a process, say p_i, cannot block until some specific interaction, say I, is enabled. This follows because interaction I may never be enabled and some other interaction of which p_i is a member may be enabled.

A large number of algorithms have been devised to implement binary and multiway interactions [Sch82, BS83, Sis84, Bag89b, CM88, Ram87, Bag89a]. Three primary classes of properties are associated with such algorithms: safety, liveness, and fairness. Although most existing algorithms satisfy the safety and liveness properties, few implement fairness. In [Fra86] Francez gives an extensive overview of fairness notions and demonstrates the effects of some of them on program correctness.

In this paper, we specify the general problem of implementing interactions and formally express the desired safety, liveness, and fairness properties. Two types of strong fairness are considered: Strong Interaction Fairness (SIF), which requires that if an interaction is enabled infinitely often, it be started infinitely often, and Strong Process Fairness (SPF), which requires that if a process is ready to participate in some enabled interaction infinitely often, it does so infinitely often. It has been shown [TB89] that SIF is impossible for binary (and hence for multiway) interactions and

SPF is impossible for multiway interactions.

We describe an efficient algorithm that satisfies the requirements of SPF for binary interactions. This algorithm is a significant improvement on existing algorithms: other than Sistla [Sis84], none of the existing algorithms provide strong fairness. Sistla's algorithm guarantees strong fairness, but has two drawbacks. First, it assumes that every process will eventually become idle. Apart from limiting the applicability of this algorithm, this assumption introduces unnecessary blocking and degrades performance. Secondly, the complexity of Sistla's algorithm depends on the number of processes in the system and may be affected considerably by the average time some process remains in the active state. The complexity of our algorithm is independent of both these factors and depends only on the maximum number of communicating partners for a process. In graph theoretic term, whereas the complexity of the former algorithm is determined by the size of the graph, ours is dependent only on its degree.

The remainder of the paper is organized as follows: Section 2 gives a brief description of our computation model. Section 3 gives a formal specification of the problem and the properties desired in the solution. Section 4 uses the notion of knowledge in a distributed system [CM86] to derive the necessary and sufficiet condition that must be satisfied by any algorithm that solves the fair synchronization problem. Section 5 contains the algorithm, its correctness proof, and its message and time complexities. Section 6 discusses the optimality of our algorithm.

2 Model and Definitions

A program is essentially a set of variables and a set of state transition rules. The state of a program consists of the values assumed by its variables. A computation of a program starts from an initial state and goes on forever; in each step of the computation, a rule is selected nondeterministically for execution. Each computation uniquely determines an infinite sequence of program states. The meaning of a program is thus characterized by the set of all possible computations. With this view of program, conventional temporal logics [Krö87, ES89] can be adopted to describe properties of programs. We omit the definition of our logical language; an explanatory statement will be provided along with every formally stated property of a program.

Programs (program modules) can be combined to produce composite programs in a natural way. The set of rules of a composite program is the union of those of its constituting modules. A program composed of programs F and G is denoted by $F[\!]G$; F and G may themselves be composite programs. Variables belonging to more than one modules are termed *shared* variables. Program modules communicate with one another by shared variables. We are interested in distributed programs where program modules are functionally divided into two categories: *processes* which do

significant computations and *channels* which simply relay messages. Messages are modeled as the values written to and read from the shared variables between a process and a channel.

A more detailed description of our model can be found in [TB89].

3 Problem Specification

We adapt the UNITY format of problem specification [CM88]: Let *USER* refer to a set of asynchronous processes (including the channels) and *OS* refer to the (distributed) scheduler that implements synchronizations among the asynchronous processes in *USER*. The composite program $USER[]OS$ is referred to as $\mathcal{P}$. We use the temporal logic language introduced in section 2 to specify the properties of *USER* and $\mathcal{P}$ as well as the constraints on *OS*.

We assume processes in *USER* are numbered 1 through n and the i-th process is denoted by $user_i$; analogously for *OS*. $p_i \equiv user_i[]os_i$ denotes the i-th process in $\mathcal{P}$. We shall refer to a process in *USER* as a *user*, a process in *OS* as an *os*, and a process in $\mathcal{P}$ as a *process*. An interaction among $user_i$, $user_j$, and $user_k$ is represented by $\{i,j,k\}$. $\mathcal{I}$ is the set of interactions defined among users; each element of $\mathcal{I}$ is a nonempty subset of $\{1,2,...,n\}$. Two interactions are said to be *conflicting* if they have at least one common member. The set of all interactions of which a process is a member is referred to as the interaction set of the process.

Each *user* and the corresponding *os* share two variables: a boolean array called **flag** and variable **state** which may assume the value *active*, *idle*, or *commit*. The three states of a process correspond to a user that does not want to participate in any interaction, a user that is waiting to participate in some interaction, and a user that has committed itself to a specific interaction. Each component of **flag** corresponds to an interaction in the user's interaction set. Interaction I is started if one of its members, say p_i, sets $\mathbf{flag}_i^I$ to 1 and is terminated if $\mathbf{flag}^I$ is set back to 0 for all members of I. We introduce some abbreviations for commonly used predicates:

$$active_i \equiv (\mathbf{state}_i = active), \text{ similary for } idle_i \text{ and } commit_i \quad \text{(d1)}$$

$$enable^I \equiv (\forall i : i \in I :: idle_i) \quad \text{(d2)}$$

$$sync^I \equiv (\exists i : i \in I :: \mathbf{flag}_i^I = 1) \quad \text{(d3)}$$

$$E[I,J] \equiv (I \neq J \wedge I \cap J \neq \phi) \quad \text{(d4)}$$

All assertions are assumed to be universally quantified over all values of their free variables. In the remainder of this section, we formalize the behavior of a user process, the constraints imposed on the scheduler and the safety, liveness, and fairess properties desired in the composite program.

3.1 Specification of *USER*

This part specifies the behavior of the *USER* program at its interface with the *OS* and also specifies some properties that are guaranteed when *USER* is composed with the *OS*.

For each user, **state** is initialized to *active* and each component of **flag** to 0. An active user may autonomously transit to idle — (u1). An idle user may transit to commit only after some interaction in its interaction set is started — (u2) and the user transits from commit back to active only after the interaction is terminated — (u4). When an interaction is started, all members will eventually transit to commit — (u3). A user may not start an interaction — (u5).

$active_i$ *Unless* $idle_i$ *in USER* (u1)

$idle_i$ *Unless* $commit_i \wedge (\exists I : i \in I :: sync^I)$ *in USER* (u2)

$sync^I \rightarrow \Diamond(\forall i : i \in I :: commit_i)$ *in* $\mathcal{P}$ (u3)

$commit_i$ *Unless* $active_i \wedge (\forall I : i \in I :: \neg sync^I)$ *in USER* (u4)

$(\mathbf{flag}_i^I = 0) \rightarrow \bigcirc(\mathbf{flag}_i^I = 0)$ *in USER* (u5)

3.2 Specification of $\mathcal{P}$

This part specifies the safety and liveness properties that must be provided by the composition of *USER* and *OS*.

The safety properties require that only *enabled* interactions can be started — (pp1) and that conflicting interactions cannot be started simultaneously — (pp2). The liveness property requires that if an interaction I is *enabled*, either I or a conflicting interaction be eventually started — (pp3).

$\neg sync^I$ *Unless* $enable^I$ *in* $\mathcal{P}$ (pp1)

$E[I, J] \rightarrow \neg(sync^I \wedge sync^J)$ *in* $\mathcal{P}$ (pp2)

$enable^I \rightarrow \Diamond(sync^I \vee (\exists J : E[I, J] :: sync^J))$ *in* $\mathcal{P}$ (pp3)

(pp3) and (u3) imply that an enabled interaction will eventually be disabled.

3.3 Constraints on *OS*

The only shared variables between $user_i$ and os_i are $\mathbf{state}_i$ and $\mathbf{flag}_i$. For each os, **state** is initialized to *active* and each component of **flag** to 0 (consistent with the initialization in *USER*). An os may not change the state of a user so that the properties (u1), (u2), and (u4) are preserved in the composite program $\mathcal{P}$. Furthermore, an os may not terminate an interaction.

3.4 Fairness

The notion of fairness in the problem specification is weak. A user is said to be *ready* if some interaction in its interaction set is enabled. The problem specification allows starvation: from a

certain point of computation, a user may become ready infinitely many times but never participate in any interaction. Also, an interaction may be enabled infinitely many times but never be started.

We are interested in two stronger fairness notions: Strong Process Fairness (SPF) and Strong Interaction Fairness (SIF). SPF asserts a user that is infinitely often ready will infinitely often participate in some interaction. SIF asserts an interaction that is enabled infinitely often will be started infinitely often. It can be shown that SPF subsumes (pp3).

$ready_i \equiv (\exists I : i \in I :: enable^I)$

SPF $\equiv \Box\Diamond ready_i \rightarrow \Box\Diamond(\exists I : i \in I :: sync^I)$

SIF $\equiv \Box\Diamond enable^I \rightarrow \Box\Diamond sync^I$

We call an additional property *satisfiable* if there exists an *OS* that satisfies the original specification such that the additional property also holds; otherwise it is *unsatisfiable.*

It has been shown [TB89] that SIF is unsatisfiable for the binary (and hence multiway) interaction problem and SPF is unsatisfiable for the multiway interaction problem.

4 Derivation of a Fair Algorithm

We describe an algorithm for the binary interaction problem with SPF. The design process of the algorithm is elaborated using reasoning about knowledge.

4.1 Knowledge

Suppose s is a program state of $\mathcal{P}$. $s[p_i]$ denotes the projection of s on p_i, i.e. the state of p_i at s. Let b be a predicate on program states. The knowledge of p_i about b at s, $K_i b$ *at* s, is defined as $\forall s' : s' \in Reach(\mathcal{P}) \wedge s'[p_i] = s[p_i] :: b$ *at* s', where $Reach(\mathcal{P})$ is the set of states reachable from an initial state. Note that $K_i b \rightarrow b$. A predicate is *local* to p_i if its truth value can always be determined by the state of p_i. So, if b is local to p_i, then it is always the case that $K_i b$ or $K_i \neg b$.

The transfer of knowledge among processes in a distributed system is well explored in [CM86]. We shall refer to the results as principles of knowledge transfer. Of particular interest to us are the following properties: A process p_i may not have the knowledge about a local predicate of another process p_j unless p_i receives a message (from p_j). Conversely, p_i may not lose the knowledge about a local predicate of p_j unless p_i sends a message (to p_j).

4.2 (pp1)–(pp3) and SPF Revisited

In this section, we derive the necessary and sufficient conditions that must be satisfied by any solution to the binary interaction problem with SPF. We informally argue their correctness and formally prove them in the appendix.

To start an interaction, say $\{i,j\}$, either p_i or p_j has to set the corresponding **flag** ($\mathbf{flag}_i^{\{i,j\}}$ or $\mathbf{flag}_j^{\{i,j\}}$) to 1. As stipulated by the safety properties (pp1) and (pp2), a process that starts an interaction must have some knowledge about the local states of its communicating partners. Let $\tilde{K}_i^{\{i,j\}}$ ($\tilde{K}_j^{\{i,j\}}$) represent the state of knowledge needed by p_i (p_j) before it may start interaction $\{i,j\}$. As either p_i or p_j may start interaction $\{i,j\}$, we define $\tilde{K}^{\{i,j\}} \equiv \tilde{K}_i^{\{i,j\}} \vee \tilde{K}_j^{\{i,j\}}$. The following assertion simply states that $\tilde{K}^{\{i,j\}}$ must necessarily be achieved before interaction $\{i,j\}$ can be started.

$$\neg sync^{\{i,j\}} \wedge \bigcirc sync^{\{i,j\}} \rightarrow \tilde{K}^{\{i,j\}}. \tag{kp1}$$

To ensure that a process starts at most one interaction, (kp1) must be strengthened by the following assertion:

$$\neg(\exists j,k : j \neq k :: (\mathbf{flag}_i^{\{i,j\}} = 1) \wedge (\mathbf{flag}_i^{\{i,k\}} = 1)). \tag{kp2}$$

We now consider fairness. SPF requires that if some p_i is infinitely often ready to participate in some (possibly different) interactions, it eventually succeed in doing so. Without loss of generality, assume that p_i eventually execute interaction $\{i,j\}$. Due to (kp1), interaction $\{i,j\}$ may be started only if $\tilde{K}^{\{i,j\}}$ has been achieved. This leads us to the following predicate as being necessary for SPF.

$$\Box\Diamond ready_i \rightarrow \Box\Diamond(\exists j :: \tilde{K}^{\{i,j\}}) \tag{kp3}$$

(kp3) would be sufficient for SPF, if we could argue that $\tilde{K}^{\{i,j\}} \rightarrow \Diamond sync^{\{i,j\}}$. This however is not the case: in general, a process p_i may simultaneously achieve $\tilde{K}_i^{\{i,j\}}$ and $\tilde{K}_i^{\{i,k\}}$ for some j,k such that it has to choose either $\{i,j\}$ or $\{i,k\}$ but not both. Thus (kp3) must be strengthened by the following predicate which asserts that if $\tilde{K}^{\{i,j\}}$ is achieved infinitely often, both p_i and p_j will infinitely participate in some interaction from their respective interaction sets.

$$\Box\Diamond\tilde{K}^{\{i,j\}} \rightarrow \Box\Diamond(\exists k :: sync^{\{i,k\}}) \wedge \Box\Diamond(\exists k :: sync^{\{j,k\}}) \tag{kp4}$$

We now turn to the issue of what should constitute $\tilde{K}^{\{i,j\}}$, particularly $\tilde{K}_i^{\{i,j\}}$. (pp1) requires that only enabled interactions may be started, which suggests that $\tilde{K}_i^{\{i,j\}}$ should imply $K_i\, enable^{\{i,j\}}$, i.e. $K_i(idle_i \wedge idle_j)$ or $(idle_i \wedge K_i idle_j)$. (pp2) requires that conflicting interactions cannot be started simultaneously. It follows that $\tilde{K}_i^{\{i,j\}}$ should also imply that no other interactions involving either p_i or p_j has been started; in other words $K_i((\forall k :: \neg sync^{\{j,k\}}) \wedge (\forall k :: \neg sync^{\{i,k\}}))$. As $K_i(\forall k :: \neg sync^{\{j,k\}})$ is implied by $K_i enable^{\{i,j\}}$, we define $\tilde{K}_i^{\{i,j\}} \equiv K_i enable^{\{i,j\}} \wedge K_i(\forall k :: \neg sync^{\{i,k\}})$ or $K_i(enable^{\{i,j\}} \wedge (\forall k :: \neg sync^{\{i,k\}}))$.

Theorem 1 (kp1)–(kp4) *if and only if* (pp1)–(pp3) *and* SPF. (proof in appendix, page 21)

4.3 The Fair Algorithm

We first investigate how $\tilde{K}_i^{\{i,j\}}$ can be achieved. The first conjunct of this knowledge predicate $K_i enable^{\{i,j\}}$ requires that p_i is idle and p_i knows p_j is idle. By the principles of knowledge transfer, p_i cannot know whether p_j is idle unless p_i receives a message from p_j after p_j becomes idle. Suppose p_j becomes idle at some point of computation. To help p_i achieve $K_i enable^{\{i,j\}}$, p_j sends a message to p_i. Let's call this message a *request*. When p_i receives the request, if p_i itself is idle, then $K_i enable^{\{i,j\}}$ is achieved. Note that p_j should send a request to at most one process; otherwise the requests will not help other processes achieve sufficient knowledge to start interactions.

The second conjunct $K_i(\forall k :: \neg sync^{\{i,k\}})$ requires that p_i knows that no interaction in its interaction set is started. Since a process will not start any interaction unless it receives a request, the knowledge predicate is true in any initial state. After becoming idle, if p_i does not send any request to other processes then the knowledge predicate remains true according to the principles of knowledge transfer. Even if p_i has sent some request, but has also received a "denial," then the knowledge predicate is still true. (An obvious reason for a process to deny a request is if it is not idle. We shall explore this issue in detail.) In the first case, by not sending any request p_i does not lose the knowledge $K_i(\forall k :: \neg sync^{\{i,k\}})$; while in the second case, it regains the knowledge by receiving a denial to each request it sent.

In summary, $\tilde{K}_i^{\{i,j\}}$ is achieved when p_i receives a request from p_j, and p_i is idle and does not have any outstanding request. At this moment, p_i can start interaction $\{i,j\}$. After starting the interaction, p_i has to send an *accept* message to p_j such that p_j will know $\{i,j\}$ is started. It is interesting to note that p_i may send an *accept* message to p_j without starting interaction $\{i,j\}$. On receiving the *accept* message, p_j achieves $\tilde{K}_j^{\{i,j\}}$ and can start $\{i,j\}$.

From the analysis above, to achieve $\tilde{K}^{\{i,j\}}$, either p_i or p_j or both should be "willing" to send request to the other process. To prevent a process from constantly denying another (and thus wasting a lot of messages), we adopt a token scheme [Bag89b]. We associate a unique token with each interaction; a token embodies a request. Only the process with a token may send a request, i.e. the token, to the other process of the corresponding interaction.

The problem of when a process should deny a request remains unresolved. Since a process may never become idle, it should deny any request that it receives when it is not idle. Also, a process cannot participate in more than one interactions, so it should deny any request when it has already participated in some interaction. The situation is complicated only when a process receives a request while it is waiting for a reply to its own request. To avoid deadlock, a process may deny a request whenever it has an outstanding request; however, this naive approach will cause livelock.

We introduce asymmetry by assigning a unique non-negative integer (id) to each interaction (token or request) [Bag89b]. p_i denies a request from p_j when the id of the request is smaller than that of its outstanding request to p_k; it delays the reply otherwise. In the latter case, if p_i receives a denial from p_k, it achieves $\tilde{K}_i^{\{i,j\}}$ and accepts the request from p_j; if it receives an accept from p_k, which indicates that $\tilde{K}_k^{\{i,k\}}$ was achieved, it denies the request from p_j.

With a way to achieve $\tilde{K}^{\{i,j\}}$, an algorithm can satisfy (kp1). (kp2) and (kp4) are straightforward. (kp3) demands that $\tilde{K}^{\{i,j\}}$ be achieved in a fair manner; in other words, if interactions involving some p_i are enabled infinitely often, than p_i should not be ignored infinitely often. This may be achieved by requiring that a process be allowed only a finite number of transitions to the idle state, before it is required to send a specific token owned by it.

For this purpose, each process maintains the tokens owned by it in a queue. Tokens received by the process are inserted in the queue in FIFO manner. The requirements of (kp3) can be satisfied by requiring that the process send tokens from this queue in a FIFO order. However, for the sake of efficiency, we follow a slightly different order. When a process becomes idle, if its token queue is not empty, it requests the token at the head of the queue before replying to any request. If its first request is denied, the subsequent requests will be made by sending tokens in the descending order of token id's. This mechanism ensures that (a) if a process becomes idle infinitely often, no token owned by it can be ignored indefinitely and (b) the token at the head of queue is denied at most twice while others are denied at most once, before a process participates in some interaction.

Furthermore, it is not necessary that all tokens have distinct id's. It suffices to number tokens in such a way that the id's of tokens associated with conflicting interactions are distinct. We subsequently prove that this schema has a significant impact on the time complexity of the algorithm.

5 The Algorithm and Its Correctness Proof

5.1 The *OS*

We describe the code executed by an *OS* process using the fair algorithm. We assume that an *OS* process has a separate output queue to each neighbor, but a single input queue from all neighbors; all incoming messages to this process are appended to the single input queue. The channels among the processes are defined to match this structure; a channel has a number of input queues and a single output queue. The task of a channel is to transfer messages from its input queues to its only output queue such that no message is ignored indefinitely. Since the task of a channel is extremely simple and well understood, we show only the program for a process.

VARIABLE (Variable subscripts are omitted, as is the declaration for some variables whose purpose is obvious.)

state,flag: shared with *USER*; see problem specification.

token_q: a queue of tokens. Each token is a triple (no,p_i, p_j), where no is a token identifier (id) and p_i and p_j are the members of the corresponding interaction.
Four operations — *maxdequeue*, *dequeue*, *enqueue*, and *empty* are associated with this queue, where *maxdequeue* removes and returns the token with the highest id from the queue.
dequeue: remove and return the token in the head.
enqueue: append a token to the end of queue.
empty: check whether the queue is empty.
Initially, each token is arbitrarily assigned to one of the processes named in the token.

observe: indicates that (state=*idle*) for the user has been observed by the corresponding os. Initially *false*.

pending: indicates whether the process has an outstanding request. Initially *false*.

rno: the number of the token corresponding to the outstanding request. Initially *null*.

rid: the id of the requested process. Initially *null*.

myid: the id of this process.

delay: indicates whether the process has delayed a request. Initially *false*.

delay_token: the delayed token. Initially *null*.

com: an array of boolean variables, each element of which corresponds to an element of flag. com[j] is set to *true* if the process is about to participate in interaction {myid,j}. Initially *false*.

In the following description, the expression "*receive*(msg) $\wedge$ *pred*," where msg is token, yes, no, or done and *pred* is a predicate which does not contain any *receive*, is used as a shorthand to describe the following actions: First, checks if the predicate is true, and if so determine whether the type of the message at the head of the input queue is the same as that of msg; if so, extracts the message and assigns it to msg in the beginning of the body of the rule. The rule is disabled if the type checking fails or *pred* is false. In the following description, com* denotes ($\exists j$ :: com[j]). The use of token.p in one process refers to the other process named in the token.

RULE

R1: /* Observing the transition to *idle* state by *USER* */
$\neg$com* $\wedge$ ((state = *idle*) $\wedge$ $\neg$observe) $\rightarrow$ [observe := *true*;
if $\neg$*empty*(token_q) then [Request(*true*);
pending := *true*;]]

R2: /* Requesting an interaction */
$\neg$com* $\wedge$ observe $\wedge$ $\neg$pending $\wedge$ $\neg$*empty*(token_q) $\rightarrow$ [Request(*false*);
pending := *true*;]

R3: /* Accepting a request */
R3.1: *receive*(token) $\wedge$ $\neg$com* $\wedge$ observe $\wedge$ $\neg$pending $\rightarrow$ [com[token.p] := *true*;
observe := *false*;

Accept(token);]

R3.2: *receive*(no) ∧ delay → [pending := *false*;
rno := *null*; rid := *null*;
com[token.*p*] := *true*;
observe := *false*;
Accept(delay_token);
delay := false;]

R4: /* Starting an interaction */
receive(yes) → [pending := *false*;
if delay then [Deny(delay_token);
delay := *false*;]
flag[rid] := 1;
com[rid] := *true*;
observe := *false*;]

R5: /* Refusing a request */
receive(token) ∧ (com* ∨ ¬observe ∨ delay ∨ (pending ∧ (token.*no* < rno))) → Deny(token);

R6: /* Delaying the reply to a request */
receive(token) ∧ (pending ∧ (token.*no* ≥ rno)) ∧ ¬delay → [delay_token := token;
delay := *true*;]

R7: /* Relinquishing a request */
receive(no) ∧ ¬delay → [pending := *false*;
rno := *null*; rid := *null*;]

R8: /* Detecting the termination of an interaction by *USER* */
R8.1 (rid ≠ *null*) ∧ (flag[rid] = 0) ∧ com[rid] → [com[rid] := *false*;
send *done* to os_{rid};]

R8.2 *receive*(done) → com := *false*; /* array assignment */

Request(first): if first then token := *dequeue*(token_q)
else token := *maxdequeue*(token_q);
rno := token.*no*; rid := token.*p*;
send token to os_{rid};

Deny(tk): send *no* to $os_{tk.p}$;
enqueue(token_q,tk);

Accept(tk): send *yes* to $os_{tk.p}$;
enqueue(token_q,tk);

5.2 Correctness Proof

We show that the distributed program $\mathcal{P}$ with OS using the fair algorithm satisfies (kp1)-(kp4), which were shown previously to be the necessary and sufficient conditions of (pp1)-(pp4) and SPF. The safety properties (kp1) and (kp2) are proved in theorems 2 and 3 respectively; whereas the liveness properties (kp3) and (kp4) are proved in theorems 4 and 5. It should be clear that the OS in the previous subsection meets the constraints on OS in the problem specification. Unless otherwise stated, all assertions in the lemmas, corollaries, and theorems are assertions on $\mathcal{P}$.

We assume that the channels connecting the processes deliver every message exactly once but not necessarily in order. Let in_i be the set of messages in the input queue of os_i and $out_j^{\{i,j\}}$ the set of messages in the output queue of os_j destined for os_i.

5.2.1 Safety Properties: (kp1) and (kp2)

Lemma 1 $\mathsf{pending}_i \rightarrow \neg\mathsf{com}_i^*$. *A process with an outstanding request cannot commit to any interaction.*

Proof. $\mathsf{pending}_i$ is initialized to *false*, so the statement is true initially. $\mathsf{pending}_i$ becomes true due to R1 or R2. Both rules are enabled only if $\neg\mathsf{com}_i^*$ holds and $\neg\mathsf{com}_i^*$ remains true after the execution of either rule. $\neg\mathsf{com}_i^*$ becomes false due to R3.1, R3.2, or R4. R3.1 is enabled only if $\mathsf{pending}_i$ is false and $\mathsf{pending}_i$ remains false after the execution of R3.1; while in R3.2 or R4, $\neg\mathsf{com}_i^*$ and $\mathsf{pending}_i$ become false together.

Lemma 2 $((yes \in in_i) \rightarrow \mathsf{pending}_i) \wedge ((no \in in_i) \rightarrow \mathsf{pending}_i)$.

Proof. We show that $(yes \in in_i) \rightarrow \mathsf{pending}_i$; the case for *no* can be argued analogously. From R3, a process p_j will send *yes* to another process p_i only if p_j has received a token from p_i and has not yet replied to the request. By R2, p_i set $\mathsf{pending}_i$ to true, after sending a token to p_j. $\mathsf{pending}_i$ remains true until a reply is received, according to R3.2, R4, and R7. *End of Proof.*

$\mathsf{flag}_i[j]$ in the program corresponds to $\mathsf{flag}_i^{\{i,j\}}$ in the problem specification.

Lemma 3 $(\mathsf{flag}_i[j] = 1) \rightarrow \mathsf{com}_i[j] \wedge \mathsf{com}_j[i]$.

Proof. By virtue of (u5), $\mathsf{flag}_i[j]$ is set to 1 only due to the execution of R4 of os_i. The execution of R4 of os_i sets $\mathsf{com}_i[j]$ to true. R4 of os_i is enabled only after a *yes* message is sent by os_j. According to R3.1 and R3.2 whereby os_j sent *yes* to os_i, os_j must have set $\mathsf{com}_j[i]$ to true. Subsequently, $\mathsf{com}_i[j]$ is set to false only due to R8.1 of os_i and $\mathsf{com}_j[i]$ set to false due to R8.2 of os_j. However, R8.1 and R8.2 can be enabled only after $\mathsf{flag}_i[j]$ is reset to 0 by some rule in $user_i$. *End of Proof.*

Lemma 4 $\mathsf{pending}_i \rightarrow idle_i$.

Proof. We show that $\mathsf{pending}_i \rightarrow \mathsf{observe}_i$ and $\mathsf{observe}_i \rightarrow idle_i$.

The arguments will go like those in lemma 1. The first assertion is true initially, since $\mathsf{pending}_i$ is initialized false. $\mathsf{pending}_i$ becomes true due to R1 or R2; $\mathsf{observe}_i$ is set to true in R1, while R2 is enabled only if $\mathsf{observe}_i$. $\mathsf{observe}_i$ becomes false due to R3.1, R3.2, or R4; R3.2 and R4 also set $\mathsf{pending}_i$ to false, while R3.1 is enabled only if $\mathsf{pending}_i$ is false and $\mathsf{pending}_i$ remains false after the execution of R3.1.

The second assertion is true initially. From (u2) and lemma 3, $idle_i$ *Unless* com^*. By similar arguments as in lemma 1, it can be shown that $\mathsf{com}^* \rightarrow \neg\mathsf{observe}_i$. Hence, $idle_i$ *Unless* $\neg\mathsf{observe}_i$. $\mathsf{observe}_i$ is set to true due to R1, which is enabled only if $idle_i$ is true. It follows that $\mathsf{observe}_i \rightarrow idle_i$.

End of Proof.

Lemma 5 $(\mathsf{rid}_i = j) \wedge (yes \in in_i) \rightarrow idle_j$. *The requested process is idle when a positive reply from that process is received.*

Proof. When $\mathsf{rid}_i = j$, it must be os_j that sends a reply message to os_i. This follows from (a) in the Request procedure, rid_i is set to the id of the process to which os_i sends the token, (b) initially no process has an outstanding request and from R3.2, R4, and R7, os_i will not send another token until a reply for an outstanding token is received, and (c) from R3, R4, R5, and R6, a process will not send a reply to a process unless it receives a token from that process and after doing so, the token is appended to the token queue in procedure Deny or Accept.

os_j sends a *yes* to os_i by R3.1 or R3.2; R3.1 is enabled only if $\mathsf{observe}_j$ is true which implies $idle_j$ and R3.2 is enabled only if $\mathsf{pending}_j$ is true which also implies $idle_j$, from lemma 4. In the meantime, $\mathsf{pending}_i$ is true (lemma 2), which implies $\neg\mathsf{com}_i[j]$ (from lemma 1), which in turn implies $\neg sync^{\{i,j\}}$ (from lemma 3). $sync^{\{i,j\}}$ becomes true only due to R4 in os_i. From (u2), $idle_j$ remains true unless R4 is executed.

End of Proof.

Theorem 2 $\neg sync^{\{i,j\}} \wedge \bigcirc sync^{\{i,j\}} \rightarrow \tilde{K}^{\{i,j\}}$. (kp1)

Proof. $sync^{\{i,j\}} \equiv (\mathsf{flag}_i[j] = 1) \vee (\mathsf{flag}_j[i] = 1)$. Elaborating the antecedent of the assertion, $(\mathsf{flag}_i[j] = 0 \wedge \mathsf{flag}_j[i] = 0) \wedge \bigcirc(\mathsf{flag}_i[j] = 1 \vee \mathsf{flag}_j[i] = 1)$. Without loss of generality, assume that the predicate is satisfied by having $\mathsf{flag}_i[j]$ set to 1. From the perspective of p_i, the assertion states that $(\mathsf{flag}_i[j] = 0) \wedge \bigcirc(\mathsf{flag}_i[j] = 1) \rightarrow \tilde{K}_i^{\{i,j\}}$. Recall that $\tilde{K}_i^{\{i,j\}} \equiv K_i enable^{\{i,j\}} \wedge K_i(\forall k :: \neg sync^{\{i,k\}})$. By (u5), $\mathsf{flag}_i[j]$ is set to 1 only due to R4 of os_i. We need to show that R4 is enabled only if $enable^{\{i,j\}} \wedge (\forall k :: \neg sync^{\{i,k\}})$. From lemmas 2, 4, and 5, $(yes \in in_i)$ only if $(idle_i \wedge idle_j)$, i.e. $enable^{\{i,j\}}$. From lemmas 1 and 2, $(yes \in in_i)$ only if $\neg\mathsf{com}_i^*$, which implies

$(\forall k :: (\mathsf{flag}_k[i] = 0) \wedge (\mathsf{flag}_i[k] = 0))$, i.e. $(\forall k :: \neg sync^{\{i,k\}})$, by lemma 3. The result follows from R4 is enabled only if $(yes \in in_i)$. *End of Proof.*

Lemma 6 $\neg(\exists j, k : j \neq k :: \mathsf{com}_i[j] \wedge \mathsf{com}_i[k])$.

Proof. R4 is enabled only if $\mathsf{pending}_i$ is true, which implies $\neg\mathsf{com}_i^*$, from lemma 1. The rules R3.1, R3.2, and R4, in which some element of com_i is set to true, are enabled only if $\neg\mathsf{com}_i^*$ holds. *End of Proof.*

Theorem 3 $\neg(\exists j, k : j \neq k :: (\mathsf{flag}_i[j] = 1) \wedge (\mathsf{flag}_i[k] = 1))$. (kp2)

Proof. This follows from lemmas 6 and 3. *End of Proof.*

5.2.2 Liveness Properties: (kp3) and (kp4)

Lemma 7 $(msg \in in_i) \rightarrow \Diamond(msg \notin in_i)$. *The message in the input queue of os_i will eventually be extracted by os_i.*

Proof. We only need to show that the message at the head of input queue of os_i will eventually be extracted by os_i. Consider the case when the message is a *token*; other cases can be argued analogously. The token can be extracted by the execution of R3.1, R5, or R6. When a token is at the head of input queue, R3.2, R4, R7, and R8.2 are disabled and will remain disabled until the token is extracted. The execution of R1, enabled or disabled, will disable itself; similarly for R2 and R8.1. Eventually only R3.1, R5, or R6 may be enabled. The truth values of the guards of these three rules can not be changed by the executions of other disabled rules. The disjunction of these guards evaluates to true. By the fair selection criterion, the token will eventually be extracted. *End of Proof.*

From the properties of channels and the preceding lemma, a message put in $out_i^{\{i,j\}}$ by os_i will eventually be put in in_j by the corresponding channel and subsequently extracted by os_j. Let $token_i^{\{i,j\}}$ be the predicate indicating that the token associated with interaction $\{i,j\}$ is in $\mathsf{token\_q}_i$. The following is a direct consequence of lemma 7.

Corollary 1 $\Box\Diamond(token_i^{\{i,j\}} \vee token_j^{\{i,j\}})$.

Lemma 8 *The reply to an outstanding request will eventually be received, or* $\mathsf{pending}_i \rightarrow \Diamond\neg\mathsf{pending}_i$.

Proof. Consider a dynamic directed graph where each node corresponds to a process and a directed edge from os_i to os_j exists if $\mathsf{delay}_j \wedge (\mathsf{delay\_token}_j.p = i)$. Since adjacent interactions have distinct

id's, according to R6, the graph is acyclic and each node has at most one incoming edge. Since the size of the graph is finite, any directed path will stop growing from certain point of computation. Consider the last node in such a path. The request from this node will be replied by the execution of either R3.1 or R5 in the requested process and the reply will eventually be received. This node will then reply the delayed request by R3.2 or R4. Inductively, each node in the path will receive the reply to its outstanding request and reply to the request delayed by itself. *End of Proof.*

Lemma 9 *If a process delays some token, some interaction in its interaction set is eventually started.* $\mathsf{delay}_i \rightarrow \Diamond(\exists I : i \in I :: sync^I)$.

Proof. By similar arguments as in lemma 1, it can be shown that $\mathsf{delay}_i \rightarrow \mathsf{pending}_i$. From lemma 1, $delay \rightarrow \Diamond\neg\mathsf{pending}_i$. $\mathsf{pending}_i$ may be set to false by R3.2, R4, or R7. The assumption the lemma, i.e. delay_i, implies that R7 cannot be enabled. It follows that eventually either R3.2 or R4 will be enabled and executed. However, the execution of either R3.2 or R4 implies that the interaction corresponding to the outstanding request or the delayed request will be started. *End of Proof.*

From (u3), $(\exists I : i \in I :: sync^I) \rightarrow \Diamond\neg idle_i$. The following corollary follows from the preceding lemma.

Corollary 2 $\Diamond\Box idle_i \rightarrow \Diamond\Box\neg\mathsf{delay}_i$.

Lemma 10 $\Diamond\Box idle_i \rightarrow \Diamond\Box\mathsf{observe}_i$.

Proof. Due to (u3), $\Diamond\Box idle_i \rightarrow \Diamond\Box\neg(\exists I : i \in I :: sync^I)$. As R3 or R4 can cause $sync^I$ to be established for some interaction, it follows that eventually R3 and R4 must remain disabled. Meanwhile com^*, if true (lemma 3), will become false due to R8.1 or R8.2 and will remain false thereafter, i.e. $\Diamond\Box idle_i \rightarrow \Diamond\Box\neg\mathsf{com}^*$. $\mathsf{observe}_i$, if false, will become true due to R1 and will remain true, since R3 and R4 are always disabled. *End of Proof.*

Lemma 11 $\Diamond\Box idle_i \rightarrow \Diamond\Box empty(\mathsf{token\_q}_i)$. *A process that remains idle forever will eventually lose all its tokens.*

Proof. From corollary 2 and lemma 10, $\Diamond\Box idle_i \rightarrow \Diamond\Box(idle_i \wedge \neg\mathsf{delay}_i) \wedge \Diamond\Box\mathsf{observe}_i$. R1 will eventually be disabled forever. Let maxid_i be the maximum id of tokens currently in $\mathsf{token\_q}_i$; if $\mathsf{token\_q}_i$ is empty, we assume that maxid_i is set to some constant smaller than the minimum id of all tokens. Suppose $\mathsf{maxid}_i = k$ at the state when R1 is enabled for the last time. (Notice again that when R1 is enabled, other rules are disabled.) com^* is always false from that state according to the arguments in lemma 10. We need to show that $(\mathsf{maxid}_i = k) \rightarrow \Diamond((\mathsf{maxid}_i < k) \vee empty(\mathsf{token\_q}_i))$.

After R1 is executed, only R2, R5 and R7 may be enabled. os_i may acquire more tokens only due to R5. Since $\neg\mathsf{com}^* \wedge \mathsf{observe}_i \wedge \neg\mathsf{delay}_i$, R5 is enabled only if $(\mathsf{pending}_i \wedge (token_i.no < \mathsf{rno}_i))$. So, any subsequently acquired token must have id less than k. *End of Proof.*

Lemma 12 $(token_i^{\{i,j\}} \wedge \Box\Diamond(\neg\mathsf{com}_i^* \wedge idle_i \wedge \neg\mathsf{observe}_i)) \rightarrow \Diamond\neg token_i^{\{i,j\}}$. *Any token owned by os_i will eventually be sent by it if R1 is enabled sufficiently often,*

Proof. R1 is enabled if $\neg\mathsf{com}_i^* \wedge idle_i \wedge \neg\mathsf{observe}_i$. Recall that if R1 is enabled, then all other rules are disabled. So, R1 will remain enabled until it is selected for execution. When R1 is enabled, it sends tokens in FIFO order; due to the repeat execution of R1, the token for interaction $\{i,j\}$ will eventually be sent. *End of Proof.*

Lemma 13 $(\Box\Diamond idle_i \wedge \Box\Diamond\neg idle_i) \rightarrow \Box\Diamond(\neg\mathsf{com}_i^* \wedge idle_i \wedge \neg\mathsf{observe}_i)$. *If a process participates in some interaction infinitely often, then R1 is enabled infinitely often.*

Proof. Assume the contrary, i.e. $(\Box\Diamond idle_i \wedge \Box\Diamond\neg idle_i) \wedge \neg\Box\Diamond(\neg\mathsf{com}_i^* \wedge idle_i \wedge \neg\mathsf{observe}_i)$, which is evaluated to $(\Box\Diamond idle_i \wedge \Box\Diamond\neg idle_i \wedge \Diamond\Box\mathsf{com}_i^*) \vee (\Box\Diamond idle_i \wedge \Box\Diamond\neg idle_i \wedge \Diamond\Box\mathsf{observe}_i)$. com_i^* becomes false only due to R8. $\Diamond\Box\mathsf{com}_i^*$ implies that eventually some interaction in the interaction set of p_i is never terminated, which contradicts the assertion that p_i transits from *idle* to $\neg idle$ infinitely often; hence the first disjunct of the assumption is not true. $\mathsf{observe}_i$ becomes false only due to R3 and R4. $\Diamond\Box\mathsf{observe}_i$ implies that eventually no interaction in p_i's interaction set is ever started, which again violates $\Box\Diamond idle_i \wedge \Box\Diamond\neg idle_i$; hence the second disjunct is also false. *End of Proof.*

Theorem 4 $\Box\Diamond ready_i \rightarrow \Box\Diamond(\exists j :: \tilde{K}^{\{i,j\}})$. (kp3)

Proof. Assume the contrary, i.e. there exists a computation such that some p_i is infinitely often ready to participate in some interaction but $\tilde{K}^{\{i,j\}}$ is not achieved infinitely often for any interaction $\{i,j\}$ in p_i's interaction set. More formally, $\exists\sigma : \sigma \in Comp^*(\mathcal{P}) :: \Box\Diamond ready_i \wedge \neg\Box\Diamond(\exists j :: \tilde{K}^{\{i,j\}}) \mid \sigma$. Elaborate the assertion, we get $\Box\Diamond ready_i \wedge \Diamond\Box(\forall j :: \neg\tilde{K}^{\{i,j\}})$. $\Diamond\Box(\forall j :: \neg\tilde{K}^{\{i,j\}})$ implies $\Diamond\Box(\forall j :: \neg sync^{\{i,j\}})$ from theorem 2. From (u2), $\Box\Diamond ready_i \wedge \Diamond\Box(\forall j :: \neg sync^{i,j})$ implies $\Diamond\Box idle_i$, which in turn implies $\Diamond\Box empty(\mathsf{token\_q}_i)$ from lemma 11. $\Box\Diamond ready_i \equiv \Box\Diamond(\exists j : \{i,j\} \in \mathcal{I} :: idle_i \wedge idle_j)$. Since $\Diamond\Box empty(\mathsf{token\_q}_i)$, it is the case that $\neg\Diamond\Box empty(\mathsf{token\_q}_j)$ from corollary 1, which implies $\neg\Diamond\Box idle_j$ from lemma 11. $\neg\Diamond\Box idle_j = \Box\Diamond\neg idle_j$. $\Box\Diamond(idle_i \wedge idle_j) \wedge \Box\Diamond\neg idle_j$ implies $\Box\Diamond idle_j \wedge \Box\Diamond\neg idle_j$, which implies that, from lemmas 12 and 13, the token for interaction $\{i,j\}$ owned by os_j at any time will eventually be sent to os_i. This contradicts to $\Diamond\Box empty(\mathsf{token\_q}_i)$. *End of Proof.*

Theorem 5 $\Box\Diamond\tilde{K}^{\{i,j\}} \rightarrow \Box\Diamond(\exists k :: \tilde{K}^{\{i,k\}}) \wedge \Box\Diamond(\exists k :: \tilde{K}^{\{j,k\}})$. (kp4)

Proof. Assume the contrary, i.e. $\exists\sigma : \sigma \in Comp^*(\mathcal{P}) :: \Box\Diamond\tilde{K}^{\{i,j\}} \wedge \neg(\Box\Diamond(\exists k :: \tilde{K}^{\{i,k\}}) \wedge \Box\Diamond(\exists k :: \tilde{K}^{\{j,k\}}))$. The assertion is elaborated to $\Box\Diamond\tilde{K}^{\{i,j\}} \wedge (\Diamond\Box(\forall k :: \neg\tilde{K}^{\{i,k\}}) \vee \Diamond\Box(\forall k :: \neg\tilde{K}^{\{j,k\}}))$. The first conjunct implies $\Diamond\Box(idle_i \wedge idle_j)$; the second conjunct implies $\Diamond\Box idle_i \vee \Diamond\Box idle_j$. The theorem follows from a similar argument as in the previous theorem. *End of Proof.*

5.3 Complexity

The message complexity of an algorithm for the *OS* is the number of messages that a process originates or induces in the worst case, from the time the process becomes idle until it participates in some interaction. Assume that every message is delivered within one unit of time. The time complexity of an algorithm is the elapsed time in the worst case, from the time an interaction is enabled until one of its members participates in some interaction. Let D be the maximum size of interaction set.

Theorem 6 *The message complexity of the fair algorithm is* $2D + 2$.

Proof. Let q_0 be the token at the head of token queue of a process p_i that has just transit to idle and Q be the set of remaining tokens plus the tokens that may be subsequently received from other processes. Note that the id of q_0 may be smaller than that of some token in Q. Immediately after becoming idle, p_i sends q_0 by R1; assume it is denied. Subsequently, p_i will send tokens in Q in the decreasing order of their ids, according to R2. Tokens can be received only by R3.1, R5, or R6. If any token is received by the execution of R3.1 or R6, p_i will participate in some interaction by R3.1 or R2 without sending further requests. We may assume in the worst case p_i receives tokens due to R5 so that it has to keep sending tokens in Q. From R2 and R3.2, p_i has to send more tokens only if $\neg\mathsf{com}_i^* \wedge \mathsf{observe}_i \wedge \neg\mathsf{delay}_i$. As a consequence, R5 is enabled only if the received token has id smaller than that of the pending request. And since p_i sends tokens in Q in the decreasing order of their ids, it follows that each time p_i sends a token in Q, the maximum id of tokens remaining in Q will decrease. p_i has at most D tokens which are numbered distinctively, so p_i can send tokens from Q for at most D times. Each request generates at most two messages: one request and a *yes* or *no* response. Together with the 2 messages generated by the initial request q_0, this yields $2D+2$ messages. *End of Proof.*

At most $(D + 1)$ colors are needed to color the edges of a graph with maximum degree D such that adjacent edges have different colors (id's). Using edge colorings to assign token ids, the length of the longest delay chain is guaranteed to be less than or equal to $(D + 1)$, since a process delays a request only when its id is greater than that of the outstanding request, if any.

Theorem 7 *The time complexity of the fair algorithm with an edge coloring assignment of token ids is at most* $D^2 + 5D$.

Proof. Some interaction $\{i,j\}$ is enabled when both p_i and p_j are idle. From theorem 6, an idle process can generate at most $(D+1)$ requests, each of which may introduce a delay chain at most of length equal to its id. A request whose id is l will be replied within $2l$ units of time. When a request is denied, a process will continue sending tokens one by one to corresponding processes. Only the token at the head of queue may be sent twice and this token must not have the largest id; otherwise it will be delayed when it returns. Within at most $2(2+3+\ldots+D+(D+1))+2D = D^2+5D$ units of time p_i will participate in some interaction or run out of tokens without being accepted; similarly for p_j. Since it is impossible that two neighboring processes simultaneously run out of tokens without being accepted, either p_i or p_j will participate in some interaction within the amount of time. *End of Proof.*

The situation described in theorem 7 is very unlikely to happen. Assuming tokens are evenly distributed among all processes, on the average, each process owns $\frac{D}{2}$ tokens. Further, on receiving a request, the requested process will delay the request only if it has an outstanding request with a smaller id. Hence, the length of the delay chain introduced by a request will usually be much smaller than its id. Finally, according to our algorithm, the replies propagating along a delay chain will alternately be *accept* and *deny*. The three factors together imply that it is impossible for every process along a delay chain to have the worst case complexity of theorem 7. We expect that the average time complexity of our algorithm is a very small factor of the bound in theorem 7.

6 Discussion: Optimality of The Algorithm

We argue informally that the message complexity of our algorithm is within a constant difference from optimum. Also, compared to algorithms that achieve optimal message complexity, its time complexity is near optimal.

6.1 Message Bound

As has been shown in section 4, if p_i is to start interaction $\{i,j\}$, $\tilde{K}_i^{\{i,j\}}$ must be achieved. In other words, p_i must have received some message from p_j after p_j became idle so that p_i *knows* p_j is idle (which, by definition, also implies that p_j is not participating in any other interaction). We shall refer to this message as a *request* as we did in the derivation of the fair algorithm. An enabled interaction may remain enabled continuously if both members are waiting for the other to send a request, so at least one of them must "take the initiative" and eventually send a request to the other. A process is said to be *aggressive* for an interaction if it is responsible for taking

the initiative; it is *passive* for the interaction otherwise. Both members of an interaction may be aggressive simultaneously for the interaction.

We observe that, if a process receives a negative reply in response to its request, it should become passive for the corresponding interaction, while the requested process becomes aggressive. The message complexity of an algorithm that disobeys this principle is always worse than a similar algorithm that obeys the principle, since the requested process may again deny the second, third ... requests for the same reason as it denied the first. Furthermore, a process that is passive for an interaction should remain passive unless the other member of the interaction sends it a request or itself participates in some interaction. A process aggressive for an interaction should remain aggressive unless it sends a request to the other member of the interaction.

On receiving a request from p_j, p_i must eventually respond to the message; the reponse may be either positive, if p_i is idle and is not participating in any other interaction, or negative, if p_i is not ready to participate in any interaction or is participating in some interaction. Suppose p_i has D neighbors. It may happen that, at a certain point of computation, p_i is idle, not participating in any interaction, and aggressive for all interactions of which it is a member, but all its neighbors happen not to be idle. All requests from p_i will be denied by its neighbors. The total number of messages that should be charged to p_i is at least $2D$, if each request is replied directly. The preceding discussion has assumed that at most one process is aggressive for any interaction. If both processes are simultaneously aggressive for a given interaction, in the worst case, each *idle* process will still induce atr least $2D$ messages.

We consider a few common alternatives and indicate their impact on message complexity. An alternative to direct replies is that instead of replying negatively to p_j, p_i may relay the original request to another neighbor of p_j. Although, this technique may reduce the total number of messages to $D + 1$ (upto D relayed requests and 1 reply message), each message would have to be longer than in the case of direct replies and the total amount of "information bits" in this case is greater. A request typically needs at least $\log N$ bits, where N is the total number of processes, for the receiver to identify which process is the sender. (Though a token in our algorithm carries some extra information, this information can be stored within each of the two processes which share the token.) A reply needs only 1 bit. In the case of direct replies, the total number of bits that should be charged to p_i is $D(\log N + 1)$.

Using the technique of relaying requests, a request must carry information as for which interactions the requesting process is aggressive, because a process may be aggressive for any number of interactions in its interaction set. This information needs at least 1 bit and in the worst case D bits. So, the number of bits that a request carries will range from $(\log N + 1)$ to $(\log N + D)$. In

the worst case, a request will be relayed up to D times (including the original request sent by p_i) implying that the total number of bits charged to p_i is larger than $D(\log N + 1)$.

Another alternative discussed in the next section allows a passive process to send a solicitation message. The primary difference between a solicitation and a request is that a process may send multiple solicitation messages (to different processes) simultaneously. Each solicitation may induce a request message. When more than one request arrives, a process must deny all but one of them. In the worst case, a process may simultaneously receive D requests and have to deny $D - 1$ of them resulting in $3D - 1$ messages (including the D solicitation messages), which is worse than the preceding bound. Furthermore, if we assume that a process may send at most one solicitation message at any time, the algorithm is essentially the same as one which allows both processes in an interaction to be simultaneously aggressive. Once again, this modification can not improve the message complexity of the worst case to be better than 2D.

6.2 Time Bound

We consider the time complexity for deterministic algorithms that have an optimal message complexity as discussed in the previous subsection.

An idle process that has a pending request may itself receive a request from some other process. When this happens, the process must decide either to immediately deny the received request or delay its response (it definitely cannot accept the request immediately; otherwise mutual exclusion will be violated if it receives an accept for its outstanding request) in such a way that undesirable situations, e.g. deadlock or starvation, will not happen. For convenience, let the binary interaction problem be represented by an undirected graph whose nodes are processes and edges are interactions. We refer to this graph as the interaction graph. To enable a process to selectively deny or delay its response to a request, it is necessary to introduce asymmetry in some form in the interaction graph. We describe two methods to introduce asymmetry and infer a loose lower bound on the time complexity.

The first method is to *a priori* orient the interaction graph such that the resulting directed graph is acyclic. A process may request an interaction only if the orientation of the corresponding edge is outgoing. Nevertheless, a process may send a solicitation for a request along an incoming edge. Since the orientation of the graph is acyclic, a process with an outstanding request can always delay a request without introducing deadlocks. The reply to a request may be delayed by another request, which may be delayed by yet another request and so on. This delay chain can be as long as any directed path in the oriented graph. Thus every outgoing edge of a process contributes to a delay chain. However, a process p_i may also create a delay chain on an incoming edge (j,i) by sending a solicitation along that edge to p_j. If p_j has an outstanding request, it will be unable to reply to the solicitation until it receives a reply to its request. So, both a request and a solicitation

from a process may induce delay chains. The delay introduced by the delay chain is analyzed at the end of this section, following the description of a second method to orient the graph as an acyclic directed graph.

The second method is to allow a process to request any of its neighbors, while relying on a binary relation on the set of interactions to avoid deadlocks or starvation. The reaction of an idle process with an outstanding request to a request, delay or deny, is determined by the relation between the outstanding request and the received request (or equivalently between the requested process and the requesting process). To satisfy SPF, a binary relation R on $\mathcal{I}$ should be defined as follows: $(\{i,j\},\{j,k\}) \in R$ if and only if, when p_j has a pending request for interaction $\{j,k\}$ and receives a request for $\{i,j\}$ from p_i, p_j will delay its response to p_i until it receives a response from p_k to its pending request. And R must have the following two properties: First, for any pair of conflicting interactions, say I and J, either $(I,J) \in R$ or $(J,I) \in R$; otherwise starvation may happen and SPF is violated. Second, there is no cycle of length greater than 2 (e.g. (I,J), (J,K), and (K,I) will form a cycle of length 3, if I,J, and Kdo not have a common member); otherwise, the algorithm may deadlock. An edge-coloring, which we adopted for our algorithm, defines one such relation.

Let $\mathcal{R}$ denote the class of all binary relations that may be used by any algorithm ensuring SPF for binary interaction. Given some $R \in \mathcal{R}$, we define a *delay* chain as a path p_1–p_2–$\cdots$–p_l such that $(\{i,i+1\},\{i+1,i+2\}) \in R$, where $1 \leq i \leq l-2$, and the path p_1–p_2–$\cdots$–p_{l-1} does not contain any cycle. When a chain of requests is formed along p_1–p_2–$\cdots$–p_l, the request from p_1 to p_2 may be replied in $2(l-1)$ units of times but no earlier.

For a given R, we need to compute the maximum delay that can be induced on an idle process, say p_i, before it becomes active. Without loss of generality, assume p_i eventually interacts with p_j. ¿From the discussion in the previous section, p_i may request every neighbor once. p_j is the last process requested by p_i. Every p_k $(k \neq j)$ requested by p_i must eventually deny p_i's request. Each such request can establish a delay chain; however, the delay chain cannot include p_j (because, otherwise p_j must become active). Let $L^{\{i,j\}}_{\{i,k\}}(R)$ be the length of the longest possible *delay* chain starting from the edge $\{i,k\}$ without visiting p_j. $L^{\{i,j\}}_i(R) \equiv \sum_{k \neq j} L^{\{i,j\}}_{\{i,k\}}(R)$. $L_i(R) \equiv \min\{L^{\{i,j\}}_i(R)\}$. $L(R) \equiv \max_{1 \leq i \leq n}\{L_i(R)\}$.

At a certain point of computation, p_i is idle, not participating in any interaction, and aggressive for all interactions of which it is a member. Meanwhile one of its neighbors p_j, which happens to be the last one that p_i will send request to, is idle and passive for all interactions. The request from p_i to p_k, $k \neq j$, may be replied in $2L^{\{i,j\}}_{\{i,k\}}(R)$ units of time but no earlier. The total elapsed time from p_i sends out the first request until it sends a request to p_j is $2L^{\{i,j\}}_i(R)$ units. An algorithm

may at best choose p_j such that $L_i^{\{i,j\}} = L_i(R)$. The time complexity is $2L(R)$ for an algorithm based on R.

Let R^* be a relation in $\mathcal{R}$ such that $\forall R : R \in \mathcal{R} :: L(R) \geq L(R^*)$. Such a relation would then determine the lower bound on the time complexity of the problem. $L(R^*)$ is apparently $\geq$ O(D) from the analysis of message complexity. The time complexity of our algorithm is O(D^2) indicating that $L(R^*) \leq$O(D^2). We suspect that, if $\mathcal{R}$ consists of static relations, $L(R^*)$ is O(D^2).

Both aforementioned methods rely on a relation on processes or interactions (which can be realized solely by comparisons of ids) to break symmetry. Other methods, e.g. deterministic coin tossing [CV86] (which assumes binary encoding of process ids), might reduce the time complexity at the cost of a less than optimal message complexity.

Appendix: Proof of Theorem 1

Theorem 8 (1) (kp1)–(kp4) *if and only if* (pp1)–(pp3) *and* SPF.

Proof. Note that (pp3) is subsumed by SPF and can be ignored.

(only if) Since $\tilde{K}^{\{i,j\}} \rightarrow enable^{\{i,j\}}$, (kp1) implies that $\neg sync^{\{i,j\}} \wedge \neg enable^{\{i,j\}} \rightarrow \bigcirc \neg sync^{\{i,j\}}$, which implies (pp1).

Assume (pp2) does not hold, i.e. $\exists \sigma : \sigma \in Comp^*(\mathcal{P}) :: (\exists i :: (\exists j,k :: sync^{\{i,j\}} \wedge sync^{\{i,k\}})) \mid \sigma$. ($Comp^*(\mathcal{P})$ denotes the suffix closure of the set of all possible computaions of $\mathcal{P}$.) From (kp2), interactions $\{i,j\}$ and $\{i,k\}$ must be started by different processes; we assume they are p_j and p_k, and p_k starts $\{i,k\}$ while $sync^{\{i,j\}}$ is true. According to (kp1), when p_k is to start interaction $\{i,k\}$, it must be the case that $\tilde{K}_k^{\{i,k\}}$, which implies $K_k(\forall l :: \neg sync^{\{i,l\}})$, which in turn implies $(\forall l :: \neg sync^{\{i,l\}})$, contradicting to $sync^{\{i,j\}}$.

From (kp3) and (kp4), $\Box\Diamond ready_i \rightarrow (\exists j :: \Box\Diamond(\exists k :: sync^{\{i,k\}}) \wedge \Box\Diamond(\exists k :: sync^{\{j,k\}}))$. It follows that $\Box\Diamond ready_i \rightarrow \Box\Diamond(\exists k :: sync^{\{i,k\}})$, which is SPF.

(if) Assume (kp1) does not hold, i.e. $\exists \sigma : \sigma \in Comp^*(\mathcal{P}) :: \neg sync^{\{i,j\}} \wedge \bigcirc sync^{\{i,j\}} \wedge \neg \tilde{K}^{\{i,j\}} \mid \sigma$. We assume that it is p_i to start interaction $\{i,j\}$ at σ_0.

$(\mathsf{flag}_i^{\{i,j\}} = 0 \wedge \mathsf{flag}_j^{\{i,j\}} = 0) \wedge \bigcirc(\mathsf{flag}_i^{\{i,j\}} = 1) \wedge \neg \tilde{K}_i^{\{i,j\}} \mid \sigma$.

$(\mathsf{flag}_i^{\{i,j\}} = 0 \wedge \mathsf{flag}_j^{\{i,j\}} = 0) \wedge \bigcirc(\mathsf{flag}_i^{\{i,j\}} = 1) \wedge \neg K_i(enable^{\{i,j\}} \wedge (\forall k :: \neg sync^{\{i,k\}})) \mid \sigma$.

By the definition of knowledge, $\exists s : s \in Reach(\mathcal{P}) \wedge s[p_i] = \sigma_0[p_i] :: (\mathsf{flag}_i^{\{i,j\}} = 0 \wedge \mathsf{flag}_j^{\{i,j\}} = 0) \wedge \neg(enable^{\{i,j\}} \wedge (\forall k :: \neg sync^{\{i,k\}}))$ at s.

Since p_i is to start interaction $\{i,j\}$ at σ_0 and $s[p_i] = \sigma_0[p_i]$, $\exists \tau : \tau \in Comp^*(\mathcal{P}) \wedge \tau_0 = s :: (\mathsf{flag}_i^{\{i,j\}} = 0 \wedge \mathsf{flag}_j^{\{i,j\}} = 0) \wedge \bigcirc(\mathsf{flag}_i^{\{i,j\}} = 1) \wedge \neg(enable^{\{i,j\}} \wedge (\forall k :: \neg sync^{\{i,k\}})) \mid \tau$.

$((\mathsf{flag}_i^{\{i,j\}} = 0 \wedge \mathsf{flag}_j^{\{i,j\}} = 0) \wedge \bigcirc(\mathsf{flag}_i^{\{i,j\}} = 1) \wedge \neg enable^{\{i,j\}}) \vee ((\mathsf{flag}_i^{\{i,j\}} = 0 \wedge \mathsf{flag}_j^{\{i,j\}} = 0) \wedge \bigcirc(\mathsf{flag}_i^{\{i,j\}} = 1) \wedge \neg(\forall k :: \neg sync^{\{i,k\}})) \mid \tau$.

The second disjunct contradicts to (pp2). With (u1), (u5), and constraints on *OS*, (pp1) implies that $\neg sync^{\{i,j\}} \wedge \neg enable^{\{i,j\}} \rightarrow \bigcirc \neg sync^{\{i,j\}}$, which invalidates the first disjunct.

(kp2) holds trivially.

Assume (kp3) does not hold, i.e. $\exists \sigma : \sigma \in Comp^*(\mathcal{P}) :: \Box\Diamond ready_i \wedge \neg\Box\Diamond(\exists j :: \tilde{K}^{\{i,j\}}) \mid \sigma$.

$\Box\Diamond ready_i \wedge \Diamond\Box(\forall j :: \neg\tilde{K}^{\{i,j\}}) \mid \sigma$.

From (kp1), $\Diamond\Box(\forall j :: \neg\tilde{K}^{\{i,j\}}) \rightarrow \Diamond\Box(\forall j :: \neg sync^{\{i,j\}}) \mid \sigma$.

$\Box\Diamond ready_i \wedge \Diamond\Box(\forall j :: \neg sync^{\{i,j\}}) \mid \sigma$, contradicting to SPF.

By an argument similar to that for (kp3), SPF implies (kp4). Note that $\tilde{K}^{\{i,j\}} \rightarrow ready_i \wedge ready_j$.

End of Proof.

References

[Bag89a] R.L. Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering*, pages 1053–1065, September 1989.

[Bag89b] R.L. Bagrodia. Synchronization of asynchronous processes in CSP. *ACM Transactions on Programming Languages and Systems*, 11(4):585–597, October 1989.

[BS83] G. Buckley and A. Silberschatz. An effective implementation of the generalized input-output construct of CSP. *ACM Transactions on Programming Languages and Systems*, 5(2):223–235, April 1983.

[CM86] K.M. Chandy and J. Misra. How processes learn. *Distributed Computing*, 1(1):40–52, 1986.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[CV86] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 206–219, 1986.

[ES89] E.A. Emerson and J. Srinivasan. Branching time temporal logic. In J.W. de Bakker, W.P. de Roever, and Rozenberg G., editors, *LNCS 354: Linear Time, Branching Time and Partial Order in Logic and Models for Concurrency*, pages 123–172. Springer-Verlag, 1989.

[FHT86] N. Francez, B. Hailpern, and G. Taubenfeld. Script: A communication abstraction mechanism. *Science of Computer Programming*, 6(1):35–88, January 1986.

[Fra86] N. Francez. *Fairness*. Springer-Verlag, 1986.

[Fra89] N. Francez. Cooperating proofs for distributed programs with multiparty interactions. *Information Processing Letters*, 32(5):235–242, September 1989.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *CACM*, 21(8):666–677, August 1978.

[Krö87] F. Kröger. *Temporal Logic of Programs*. Springer-Verlag, 1987.

[Lyn80] N.A. Lynch. Fast allocation of nearby resources in a distributed system. In *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, April 1980.

[Ram87] S. Ramesh. A new and efficient implementation of multiprocess synchronization. *LNCS 259: PARLE Parallel Architecture and Languages Europe*, pages 387–401, June 1987.

[Sch82] F. Schneider. Synchronization in distributed programs. *ACM TOPLAS*, 4(2):125–148, April 1982.

[Sis84] A.P. Sistla. Distributed algorithms for ensuring fair interprocess communication. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pages 266–277, 1984.

[TB89] Y.-K. Tsay and R.L. Bagrodia. Some impossibility results in interprocess synchronization. Technical report, Computer Science Department, UCLA, October 1989.