ELSEVIER

Theoretical Computer Science 243 (2000) 307–338

Theoretical Computer Science

www.elsevier.com/locate/tcs

Two decentralized algorithms for strong interaction fairness for systems with unbounded speed variability

Yuh-Jzer Joung *

Department of Information Management, National Taiwan University, Taipei, Taiwan, ROC

Received September 1997; revised August 1998
Communicated by M. Nivat

Abstract

We present two randomized algorithms, one for message passing and the other for shared memory, that, with probability 1, schedule multiparty interactions in a strongly fair manner. Both algorithms improve upon a previous result by Joung and Smolka (proposed in a shared-memory model, along with a straightforward conversion to the message-passing paradigm) in the following aspects: first, processes' speeds as well as communication delays need not be bounded by any predetermined constant. Secondly, our algorithms are completely decentralized, and the shared-memory solution makes use of only single-writer variables. Finally, both algorithms are *symmetric* in the sense that all processes execute the same code, and no unique identifier is used to distinguish processes.

Keywords: Randomized algorithm; Strong interaction fairness; Weak interaction fairness; Multiparty interaction

1. Introduction

Since Hoare introduced CSP [13], *interactions* and *nondeterminism* have become two fundamental features in many programming languages for distributed computing (e.g., Ada [34], Script [11], Action Systems [3], IP [10], and DisCo [15, 14]) and algebraic models of concurrency (e.g., CCS [24], SCCS [23], LOTOS [7], π-calculus [25, 26]). Interactions serve as a synchronization and communication mechanism: the participating processes of an interaction must synchronize before embarking on any data transmission. Nondeterminism allows a process to choose one interaction to execute, from a set of potential interactions it has specified.

For example, consider a replica system consisting of two client processes C_1 and C_2, and two replica managers M_1 and M_2. The two clients C_1 and C_2 interact with

* *E-mail address*: joung@ccms.ntu.edu.tw (Y. Joung).

0304-3975/00/$ - see front matter
PII: S0304-3975(98)00224-2

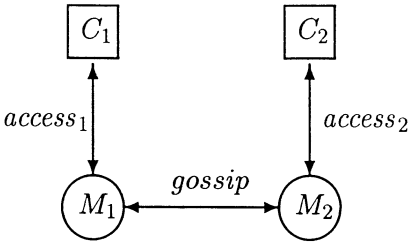

Fig. 1. A replica system.

the managers M_1 and M_2 respectively to access the database. Moreover, from time to time the two managers interact with each other to update their replica data (Fig. 1). The system can be described by the following program written in CSP's style except that input/output commands are now replaced by interactions (where $i = 1, 2$):

$C_i ::$ $*$ [$access_i \rightarrow$ *local-computing*;]

$M_i ::$ $*$ [$access_i \rightarrow$ *local-computing*;

$\square$ *gossip* $\rightarrow$ *local-computing*;]

In the program $access_i$ designates the interaction between C_i and M_i, and *gossip* designates the interaction between M_1 and M_2. Like CSP's input/output guards, interactions can also serve as guards in an alternative/repetitive command, and an interaction guard can be executed only if its participating processes are all ready for the interaction. So the replica manager M_1 can either establish an interaction with its client C_1, or an interaction with its peer M_2; and if both targets are ready, then the choice is nondeterministic. Interactions and nondeterminism therefore provide a higher level of abstraction by hiding execution-dependent synchronization activities into the implementation level.

Note that, although like CSP and Ada, each interaction in the above example involves only two processes, interactions can also be *multipartied*, allowing an arbitrary number of processes to establish an interaction. Multiparty interactions provide a higher level of abstraction than biparty interactions as they allow interactions in some applications to be naturally represented as an atomic unit. For example, the natural unit of process interactions in the famous Dining Philosophers problem involves a philosopher and its neighboring chopsticks, i.e., a three-party interaction. More examples can be found in [10], and a taxonomy of programming languages offering linguistic support for multiparty interaction is presented by Joung and Smolka [18].

Intuitively, since a process may be ready for more than one interaction at a time, the implementation of interaction guards must guarantee a certain level of fairness to avoid a prejudicial scheduling that favors a particular process or interaction. For example, the notion of *weak interaction fairness* (*WIF*) is usually imposed to ensure that an interaction that is continuously enabled will eventually be executed. (An interaction is *enabled* if its participants are all ready for the interaction, and is *disabled* otherwise.) To illustrate, the following execution of the above replica program does not satisfy WIF, as interaction $access_2$ is continuously enabled forever but is never executed (note

that, in the program, when a process is ready for interaction, it is ready to execute any interaction of which it is a member):

All four processes are ready for interaction initially, and then the following scenario is repeated forever:

- C_1 and M_1 establish $access_1$;
- C_1 and M_1 exit $access_1$ and then respectively become ready again.

WIF has been widely implemented in CSP-like biparty interactions [8, 31, 29, 5, 33], as well as in the multiparty case [28, 4, 27, 20, 17].

Although WIF can ensure some form of liveness, it is sometimes too weak to be useful. For example, consider another execution of the replica program:

All four processes are ready for interaction initially, and then the following scenario is repeated forever:

- C_1 and M_1 establish $access_1$;
- C_2 and M_2 establish $access_2$;
- the four processes respectively leave their interactions and become ready again.

The computation satisfies WIF because no interaction is continuously enabled forever. (Recall that an enabled interaction becomes disabled when some of its participants executes an interaction.) However, in the computation the two replica managers never establish an interaction, regardless of the infinitely many opportunities they have.

On the other hand, the above execution can be prevented if the implementation were to satisfy *strong interaction fairness* (*SIF*), meaning that an interaction that is infinitely often enabled is executed infinitely often. SIF is much stronger than most known fairness notions (including WIF) [2], and therefore induces more liveness properties. Unfortunately, given that (1) a process decides autonomously when it will be ready for interaction, and (2) a process's readiness for interaction can be known by another only through communications, and the time it takes two processes to communicate is nonnegligible, SIF cannot be implemented by any deterministic algorithm [32, 16]. Note that, the impossibility result holds as well even if interactions are strictly bipartied.

To cope with the impossibility phenomenon, Joung and Smolka [19] propose a randomized algorithm for scheduling multiparty interactions that guarantees SIF with probability 1. That is, if an interaction is enabled infinitely often, then the probability is 1 that it will be executed infinitely often. The algorithm is an extension of Francez and Rodeh's randomized algorithm [12] for CSP-like biparty interactions to the multiparty case. Both algorithms use a very basic idea – "attempt, wait, and check" – to establish interactions. That is, when a process is ready for interaction, it first "attempts" to establish an interaction by accessing some shared variables, and then "waits" for some Δ time before it "checks" if its partners are likewise willing to establish the interaction.[1] Francez and Rodeh were able to claim only weak interaction fairness, and only under

[1] A similar concept is used by Reif and Spirakis [30], albeit the Δ-parameter in their randomized algorithm is more deliberately calculated to meet the real-time response requirement. Like Francez and Rodeh's algorithm, however, Reif and Spirakis's algorithm is proposed only for biparty interactions, and guarantees WIF with probability 1.

the limiting assumption that the time it takes to access a shared variable (i.e., the communication delay) is negligible compared to Δ. Joung and Smolka remove the negligible delay assumption, but they require the delay be bounded by some constant ξ_{max} so that Δ can then be appropriately determined.[2] The algorithm therefore does not work for systems where such a bound cannot be known in advance. Moreover, given that the algorithm's time complexity is in linear proportion to Δ, the performance may be significantly decreased if the average communication delay is much less than the upper bound ξ_{max}.

Moreover, like Francez and Rodeh's algorithm, Joung and Smolka's algorithm is presented in a shared-memory model where processes communicate by reading from and writing to shared variables. They also have to use a multi-writer variable (meaning that a shared variable can be read and written by more than one process) for each interaction in order to resolve the mutual exclusion and concurrency problem between the participating processes of the interaction. While it is true that multi-writer variables can be implemented from single-writer variables (where a single-writer variable allows only one process to write),[3] some extra cost in efficiency would be required in the conversion.

The main contributions of this paper are two randomized algorithms for the interaction scheduling problem, one for message passing and the other for shared memory. Like Joung and Smolka's algorithm, our algorithms are presented for a multiparty setting, and use the concept of "attempt, wait, and check" to establish interactions. However, we do not assume any *predetermined* bound on the length of each process step, where a *step* is a non-zero finite time interval in which a single instruction is instantaneously executed at the last moment of the interval. (A process's *speed* is a measure of the process's steps such that the slower the speed, the more the time it takes to execute a step.) Rather, a process's Δ parameter is dynamically adjusted according to other processes' speeds. Therefore, the system's performance is determined by the actual speeds of the processes, not by a worst-case scenario of the system. We show that our algorithm guarantees SIF with probability 1, so long as the following two conditions are satisfied: (A1) processes are not hanging (a process is *hanging* if it stops executing its instructions, or there exist an infinite sequence of steps of the process with monotonically increasing length),[4] and (A2) a process's transition to a state ready for interaction does not depend on the random choices performed by other processes. Note that, the no-hanging assumption implies that the length of each process's step will eventually be bounded throughout an infinite computation of the system. However, unlike Joung and Smolka's algorithm, this bound may vary from computations to computations and, therefore, no fixed bound is assumed for all possible computations of the system.

[2] As noted by Joung and Smolka [19], the impossibility result for SIF holds as well even if the communication delay is bounded by some constant.

[3] For references on the related issues, see the book *Distributed Algorithms* by Lynch [22].

[4] A similar showdown situation has been addressed by Afek et al. [1] in solving the sequence transmission problem in an unreliable packet-switching network.

Our algorithms are completely decentralized, meaning that no coordinating process is used in either of them. In particular, for the shared-memory algorithm, only single-writer variables have been used, as opposed to Joung and Smolka's algorithm for which a multi-writer variable has to be associated with each interaction. Our algorithms are also *symmetric* in the sense that all processes execute the same code, and no unique identifiers are used to distinguish processes. Symmetry is particular useful if we are to extend the algorithms to an environment where processes can be dynamically created and destroyed. Joung and Smolka have also described how to convert their algorithm into a message-passing paradigm. However, this conversion would also turn the algorithm into asymmetric because some processes are distinguished from the others to maintain the multi-writer variables they have used in their algorithm.

To help understand our algorithms, we have chosen to present the message-passing solution first. The algorithm is simpler because a communication imposes a causal ordering between the initiator (usually the information provider) and its target (the information recipient), and the **send** and **receive** commands in the message-passing paradigm implicitly assumes this causal ordering in their executions. By contrast, a more sophisticated technique is required in a completely decentralized shared-memory model to ensure that two asynchronous processes engaged in a communication are appropriately synchronized so that the information provider will not overwrite the information before the other process has observed the content. Both algorithms share the same idea in the dynamic adjustment of the Δ-parameter.

The rest of the paper is organized as follows. Section 2 presents the multiparty interaction scheduling problem. The message-passing solution is presented in Section 3, and the shared-memory solution in Section 4. Concluding remarks are offered in Section 5.

2. The problem

We assume a fixed set of sequential processes $p_1, \ldots, p_n$ which interact by engaging in multiparty interactions $X_1, \ldots, X_m$. Each multiparty interaction X_i involves a fixed set of processes $P(X_i)$. Initially, each process in the system is in its *local computing phase* which does not involve any interaction with other processes. From time to time, a process becomes ready for a set of potential interactions of which it is a member. After executing any one of the potential interactions the process returns to its local computing phase.

Assume that a process starting an interaction will not complete the interaction until all other participants have started the interaction. Assume further that a process will eventually complete an interaction if all other participants have started the interaction. The *multiparty interaction scheduling* problem is to devise an algorithm to schedule interactions satisfying the following requirements:

Synchronization: If a process p_i starts X, then all other processes in $P(X)$ will eventually start X. Note that by the above two assumptions that a process will not complete an interaction until all other participants have started the interaction, and that

a process will eventually complete an interaction if all other participants have started the interaction, the synchronization requirement implies that when a process starts X, all participants of X will eventually complete an instance of X.

Exclusion: No two interactions can be in execution simultaneously if they have a common member. An interaction X is *in execution* if every process in $P(X)$ has started X, but none of them has yet completed its execution of X.

Strong interaction fairness: If an interaction is enabled infinitely often, then it will be executed infinitely often. (Recall that an interaction is *enabled* if its participants are all ready for the interaction, and becomes *disabled* when some of them starts an interaction.)

3. A message-passing solution

3.1. The algorithm

We now present our solution for the multiparty interaction scheduling problem in the message-passing paradigm. To help explain our algorithm, we first present a simplified version of the algorithm, which satisfies the synchronization and exclusion requirements of the problem, but does not satisfy strong interaction fairness unless the length of a process step is bounded by some predetermined constant. The restriction will be lifted later when we present the full version of the algorithm.

In the simplified version of the algorithm, each process p_i is associated with a unique token T_i. When p_i is ready for interaction, it randomly chooses one interaction X from the set of potential interactions it is willing to execute, and informs each process in $P(X)$ of p_i's interest in executing X. To do so, p_i makes $|P(X)|$ copies of T_i, tags them with "X", and sends one copy to each participant of X (including p_i itself). When all of the recipients have acknowledged the receipt of T_i, p_i waits for some Δ time, hoping that every other process in $P(X)$ will also send p_i a copy of its token tagged with "X" in this time interval.

If for each $p_j \in P(X)$, p_i does receive a copy of T_j, and each copy is tagged with "X", then p_i has successfully observed the establishment of X (because the processes in $P(X)$ all agree to execute X). Then p_i changes the tags of the tokens to "*success*". When Δ expires, p_i retrieves its tokens from each $p_j \in P(X)$ by sending p_j a message *request*, and then starts X when the tokens are returned. (Note that p_i will also receive a copy of T_i tagged with "*success*" from itself.)

If p_i does not receive copies of tokens tagged with "X" from all processes in $P(X)$ before Δ expires, then p_i also retrieves its tokens by sending each p_j a message *request*. When the tokens are returned, p_i checks if any one of them is tagged with "*success*". If so, then the process returning this token has observed the establishment of X. So p_i also starts X. If none of the tokens is tagged with "*success*", then p_i must give up on X, discard all duplicated copies of T_i, and return to the beginning of this procedure to attempt another interaction.

```
1  *[ ¬ready →  do local computations; ready := true;
2  □ ready ∧ ¬commit ∧ attempt = nil →
3        randomly select an interaction X for which p_i is ready;
4        attempt := X;
5        send a copy of T_i tagged with "X" to each p_j ∈ P(X);
6        wait until each p_j ∈ P(X) acknowledges the receipt of the token;
7        init_ck := clock(p_i); /* start timing Δ */
8  □ receive T_j from p_j →
9        add T_j to token_pool;
10       send an acknowledgment to p_j;
11       ∀p_j ∈ P(attempt) : T_j ∈ token_pool ∧ tag(T_j) = attempt →
12             for each such T_j, tag(T_j) := success;
13 □ receive request from p_j →
14       remove T_j from token_pool and send it back to p_j;
15 □ clock(p_i) − init_ck ⩾ Δ →/* Δ expires */
16       send each p_j ∈ P(attempt) a message request;
17       wait until each p_j returns its copy of T_i;
18       if any returned T_i is tagged with success
19             then commit := true;
20             else attempt := nil;
21       delete the returned tokens;
22       init_ck := ∞;
23 □ commit →
24       execute attempt;
25       attempt := nil;
26       commit := false;
27       ready := false;
28 ]
```

Fig. 2. An algorithm for multiparty-interaction scheduling that may not guarantee strong interaction fairness if the length of a process step is unbounded.

The algorithm to be executed by each p_i is given in Fig. 2 as a CSP-like repetitive command consisting of guarded commands. Each guarded command is of the form "b; *message* $\rightarrow S$". A guarded command can be executed only if it is *enabled*; i.e., its boolean guard b evaluates to true and the specified message has arrived. Both the boolean guard and the message guard are optional. The execution receives the message and then the command S is executed. If there is more than one enabled guarded command, then one of them is chosen for execution, and the choice is nondeterministic. We do, however, require that a guarded command that is continuously enabled be executed eventually.

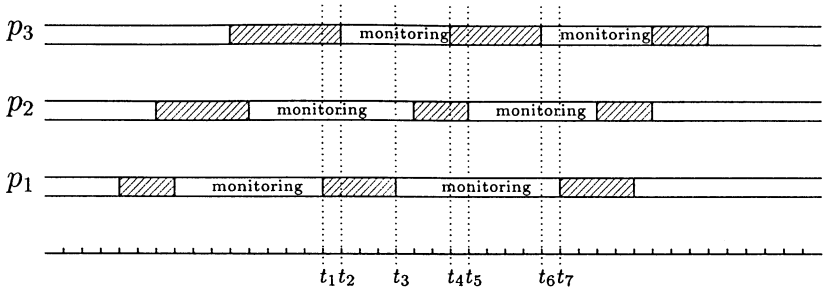

Fig. 3. A scenario of three processes executing the algorithm. Each non-shaded interval represents the time during which a process is monitoring an interaction.

The variables local to each p_i are given as follows:

- *ready*: a boolean flag indicating if p_i is ready for interaction. It is initialized to false.
- *attempt*: the interaction that p_i randomly chooses to attempt; it is set to *nil* if there is none. The initial value of *attempt* is *nil*.
- *commit*: a boolean flag indicating if p_i has committed to an interaction. It is initialized to *nil*.
- *token_pool*: set of tokens received by p_i. It is initialized to $\emptyset$.
- T_i: p_i's token. Function $tag(T_i)$ returns the tag associated with T_i.
- *init_ck*: a temporary variable for p_i to record the time at which it starts waiting for a Δ-interval before it determines whether or not its chosen interaction is established. It is initialized to ∞.

Moreover, each process p_i is equipped with a clock, and $clock(p_i)$ returns the content of the clock when the function is executed. We assume that processes' clocks tick at the same rate. Section 5 discusses how this assumption can be lifted from the algorithm.

From the above description, it is not difficult to see that the algorithm satisfies the synchronization requirement of the multiparty interaction scheduling problem (see Theorem 1). This is because a process can start an interaction X only if it has received a copy of it's token tagged with "*success*". Since only the process p_k which possesses a set of tokens $\{T_j \mid p_j \in P(X),\ tag(T_j) = \text{"}X\text{"}\}$ can change the tags to "*success*", when a process p_j finds that the token returned by p_k is tagged with "*success*", all other processes in $P(X)$ will also find that their tokens are tagged with "*success*" when they retrieve their tokens from p_k, and so will all start X. Moreover, the exclusion requirement is easily satisfied because a process attempts one interaction at a time.

The fairness property depends on an appropriate choice of Δ, however. To see this, assume that interaction X involves p_1, p_2, and p_3, which are all ready for X. We say that a process is *monitoring* X if it, after choosing X, has set up *init_ck* (line 7 of Fig. 2) and is waiting for its Δ-interval to expire (i.e., to execute line 15). Consider the scenario depicted in Fig. 3. In this figure, each non-shaded interval represents the time during which a process is monitoring an interaction. A shaded interval then

amounts to the maximum time a process can spend from the time it has executed line 15 until the time it loops back to line 7 to set a new *init_ck* to monitor another interaction. According to this scenario, p_1 is monitoring some interaction from t_3 to t_7. During this interval, p_2 and p_3 will also start monitoring some interaction (at t_5 and t_6, respectively). If the three processes monitor the same interaction, say X, then by t_5, p_1 will have received p_2's token tagged with X,[5] and by t_6, p_1 will also have received p_3's token with the same tag. So, by t_6, p_1 will have collected all three processes' tokens tagged with "X" (p_1's own token is received prior to t_3). So each process, upon receiving its own token returned by p_1, will start X.

On the other hand, if a process does not monitor an interaction long enough, then no interaction may be established among processes even if their random choices coincide. For example, consider again Fig. 3. At time t_1, p_1 has collected tokens from p_1 and p_2 (assume that they both choose the same interaction X to monitor). Suppose p_3 also chooses X to monitor at t_2. However, p_3's token is not guaranteed to arrive at p_1 before t_1, and so p_1 may give up on X at t_1 when its Δ-interval expires.

From the above discussion it can be seen that if there exists a time instance at which all processes in $P(X)$ are monitoring X, then X will be established after the processes finish up their monitoring phases. Moreover, suppose that the maximum possible interval during which each $p_i \in P(X)$ is ready for interaction but is not monitoring any interaction (i.e., the maximum possible length of a shaded interval in Fig. 3; we shall henceforth refer to each such interval as a "non-monitoring window", see Section 3.2) is strictly less than η_i. Suppose further that the processes in $P(X)$ establish their non-monitoring windows, one after another, in the following manner (assume that $P(X)=\{p_1, p_2, \ldots, p_l\}$): p_1's window is $[t, t+\eta_1-\varepsilon)$ (where the window is taken to be semi-closed because p_1 stops monitoring an interaction at t, and *starts* monitoring a new interaction at $t+\eta_1-\varepsilon$), p_2's window is $[t+\eta_1-\varepsilon, t+\eta_1+\eta_2-2\varepsilon)$, and so on. Then, we see that, at no time instance in $[t, t+\sum_{p_k\in P(X)}\eta_k - l\varepsilon)$, the processes in $P(X)$ can be all monitoring an interaction simultaneously. However, if each p_i's Δ satisfies the condition: $\Delta \geqslant \sum_{p_k\in P(X)-\{p_i\}}\eta_k$, then the processes in $P(X)$ are all monitoring an interaction at $t+\sum_{p_k\in P(X)}\eta_k - l\varepsilon$. Note that, on the condition that each p_i's Δ is greater than or equal to $\sum_{p_k\in P(X)-\{p_i\}}\eta_k$, the layout of non-monitoring windows described above provides a maximal interval throughout which we cannot find a time instance at which the processes in $P(X)$ are all monitoring an interaction.

By the algorithm, when a process is monitoring an interaction, the interaction it is monitoring is determined by the random draw performed prior to the monitoring phase. So when the processes of $P(X)$ are all monitoring interactions, the probability that X will be established after the monitoring phases is given by the probability that a set of random draws, one by each process in $P(X)$, yield the same outcome X. The *Law of Large Numbers* in probability theory (see, for example, the book by Chung [9]) then tells us that if there are infinitely many points at which all processes in $P(X)$

[5] Recall that p_2's token sent to p_1 is acknowledged by p_1, and p_2 will not start monitoring an interaction until its tokens are received by all receivers.

are monitoring interactions, then the probability is 1 that they will monitor the same interaction X infinitely often and, so, with probability 1 they will establish X infinitely often.

So, strong fairness of the algorithm relies on the assumption that the length of each non-monitoring window is bounded by some η_k so that another process's Δ can be determined accordingly. Note that the condition $\Delta \geqslant \sum_{p_k \in P(X)-\{p_i\}} \eta_k$ for p_i implies that the Δ values chosen by processes need not be the same. Moreover, a temporarily short Δ cannot cause the algorithm to err, although it may cause a set of processes to miss a chance for rendezvous.

Based on these observations, we can remove the bounded step assumption by letting processes communicate with each other about the length of their previous non-monitoring windows. Processes then use this information to adjust their next Δ-intervals. So long as processes are not hanging and every message will eventually be delivered, the dynamic adjustment of processes' Δ-intervals guarantees that when the participants of X are all ready for X, eventually their Δ-intervals will be adjusted to meet the rendezvous requirement (i.e., they will all monitor interactions at the same time). The chance that they will establish X is then determined by their random draws. In this regard, we need not assume any predetermined bound on processes' steps (speeds) and communication delays; the algorithm will adapt itself to the run-time environment.

So, we can modify the algorithm, yielding that shown in Fig. 4 — the full version of our algorithm for the multiparty interaction scheduling problem. We shall refer to the algorithm as TB (for Token-Based). Algorithm TB adds the following time variables to each p_i:

- η: records the maximum of the durations from the time p_i previously stopped monitoring interaction to the time p_i starts monitoring interaction. It is initialized to 0.
- $init\_\eta$: a temporary variable for p_i to record the time at which it starts to measure η. It is initialized to ∞.
- $E[1..n]$: $E[j]$, initialized to 0, records the maximum value of p_j's η sent by p_j.

In the algorithm, p_i measures its η by lines 1.1 and 7.1 (for the first non-monitoring window while p_i is ready for interaction), and by lines 15.1 and 7.1 (for the remaining non-monitoring windows). When p_i has sent out its token to p_j (line 5), p_j acknowledges the receipt of the token by sending its η to p_i (line 10′). Then p_i adjusts its $E[j]$ to the larger value of $E[j]$ and p_j's new η (lines 6.1–6.2). These $E[j]$'s are used in line 15′ to time-out p_i's Δ-interval.

The system's performance depends on the lengths of Δ-intervals the processes choose, which in turn depend on the values of $E[j]$'s. From time to time, one may reset each $E[j]$ (and η) after p_i has established an interaction to prevent the system getting too slow due to some abnormal speed retardation. (Note that the time variables cannot be reset while p_i is attempting to establish an interaction; for, otherwise, the algorithm would not even guarantee weak interaction fairness.) In general, since a temporarily short Δ-interval cannot cause the algorithm to err, $E[j]$ can be reset to any value, e.g., the average of the past history of $E[j]$'s values, or the minimum of them. On the other

```
1     *[ ¬ready →  do local computations;
1.1         init_η := clock(p_i); /* start measuring η */
1.2         ready := true;
2     □ ready ∧ ¬commit ∧ attempt = nil →
3           randomly select an interaction X for which p_i is ready;
4           attempt := X;
5           send a copy of T_i tagged with "X" to each p_j ∈ P(X);
6           wait until each p_j ∈ P(X) acknowledges the receipt of the token;
6.1         let η_j be the timestamp in p_j's acknowledgment;
6.2         ∀p_j ∈ P(X) − {p_i} : E[j] := max(E[j], η_j);
7           init_ck := clock(p_i); /* start timing Δ */
            /* start monitoring interaction */
7.1         η := max(η, clock(p_i) − init_η); /* record a new η */
8     □ receive T_j from p_j →
9           add T_j to token_pool;
10′         send an acknowledgment with timestamp η to p_j;
11          ∀p_j ∈ P(attempt) : T_j ∈ token_pool ∧ tag(T_j) = attempt →
12                for each such T_j, tag(T_j) := success;
13    □ receive request from p_j →
14          remove T_j from token_pool and send it back to p_j;
15′   □ clock(p_i) − init_ck ⩾ Δ, where Δ = Σ_{p_j ∈ P(attempt) − {p_i}} E[j] →
            /* Δ expires */
15.1        init_η := clock(p_i); /* start measuring η */
            /* stop monitoring interaction */
16          send each p_j ∈ P(attempt) a request;
17          wait until each p_j returns its copy of T_i;
18          if any returned T_i is tagged with success
19                then commit := true;
20                else attempt := nil;
21          delete the returned tokens;
22          init_ck := ∞;
23    □ commit →
24          execute attempt;
25          attempt := nil;
26          commit := false;
27          ready := false;
28    ]
```

Fig. 4. Algorithm TB.

hand, reseting $E[j]$'s may also bring an extra load to a stable system. This is because if $E[j]$ is reset to a value smaller than the length of p_j's next non-monitoring window then, when next time p_i wishes to establish an interaction with p_j, it may not be able to do so because p_i's Δ is too short. Therefore, extra attempts by p_i are needed for p_i to re-catch the length of p_j's non-monitoring windows. This overhead will be analyzed in Section 3.2.4.

3.2. *Analysis of algorithm* TB

In this section we prove that TB satisfies the synchronization and exclusion requirements of the multiparty interaction scheduling problem and, with probability 1, is strong interaction fair. We also analyze the expected time TB takes to schedule an interaction.

3.2.1. *Definitions*

We assume a discrete global time axis where, to an external observer, the events of the system are totally ordered.[6] Moreover, we assume that for any given time instances $t_0, t_1, \ldots$ on this axis, the usual less-than relation over these instances is well-founded. That is, for any given two time instances t_i and t_j, there are only a finite number of points $t_{i_1}, t_{i_2}, \ldots, t_{i_k}$ on the global time axis such that $t_i < t_{i_1} < t_{i_2} \cdots < t_{i_k} < t_j$. Accordingly, the phrase "there are infinitely many time instances" refers to the interval $[0, \infty]$.

Recall from TB that, a process p_i, after sending out its tokens to the processes in $P(X)$, must wait for Δ time before it decides whether to start or give up on X. We say that p_i *starts monitoring* X if it has executed line 7 of the algorithm to time its Δ. It stops *monitoring* X when line 15.1 is executed. Let t_1 and t_2, respectively, be the time at which these two events occur. The semi-closed interval $[t_1, t_2)$ is a *monitoring window* of p_i, and p_i is monitoring X in this window. Suppose that X fails to be established in this monitoring window, then p_i must start another monitoring window. Therefore, from the time (say t_0) p_i becomes ready for interaction until the time (say t_l) p_i stops monitoring an interaction that has been successfully established, the interval $[t_0, t_l)$ contains a sequence of monitoring windows $[t_1, t_2), [t_3, t_4), \ldots, [t_{l-1}, t_l)$. The interspersed intervals $[t_0, t_1), [t_2, t_3), \ldots, [t_{l-2}, t_{l-1})$ are called *non-monitoring windows*.[7] The *length* of a window is the difference of the two ends in the interval. Note that all non-monitoring windows and monitoring windows have a non-zero length. The monitoring window of p_i at time t refers to the monitoring window $[t_s, t_f)$ of p_i (if any) such that $t_s \leqslant t < t_f$; similarly for non-monitoring windows.

[6] As usual, an event transits a process from one state to another. If an event occurs at time t and it transits p from state s_1 to state s_2, then we say that p is in state s_1 *just before* t, and is in state s_2 *right after* t. For p's state to be defined at every time instance, we stipulate that p's state at time t is s_2 if the event occurs at time t.

[7] There is a latency between the time t_l at which p_i stops monitoring an interaction (line 15.1), until the time $t_{l'}$ at which p_i starts executing the interaction (line 24). To simplify the definition, we shall henceforth consider $[t_{l-1}, t_{l'})$ rather than $[t_{l-1}, t_l)$ as a monitoring window. As a result, we can say that, from the time p_i becomes ready for interaction until the time its executes an interaction, it spends all of its time in non-monitoring windows and monitoring windows.

Note that, if p_i is monitoring X, then every process in $P(X)$ must hold a copy of T_i with a tag "X". Moreover, recall that a process records the length of a non-monitoring window in variable η. Since a process records an η value only *after* it has started monitoring an interaction (line 7.1), the recorded value is slightly larger than the actual length. This is crucial to the correctness of Lemma 4.

If p_i is monitoring X at time t, then the choice of X must be the result of some random draw performed by p_i before t. Let D_{t,p_i} denote the event that is this random draw. We use $v(D_{t,p_i})$ to denote the outcome of the random draw. The probability that $v(D_{t,p_i})=X$ is denoted by $\psi_{p_i,X}$, and the probability is assumed to be independent of t. Moreover, assume $t_s \leqslant t_f$. We define a set $E_{t_s}^{t_f}P(X)$ of random draw events, at most one by each process p_i in $P(X)$, as follows:

- If p_i remains in a monitoring window throughout $[t_s, t_f]$, or p_i is in a monitoring window at t_s and then starts an interaction after the window terminates, then the random draw events D_{t_s,p_i} is included in $E_{t_s}^{t_f}P(X)$. With respect to $E_{t_s}^{t_f}P(X)$, process p_i is referred to as a type-M process.
- If p_i has a non-monitoring window contained[8] in $[t_s, t_f]$, then the random draw event performed in the window is included in $E_{t_s}^{t_f}P(X)$, and with respect to $E_{t_s}^{t_f}P(X)$, p_i is referred to as a type-N process. If p_i has more than one non-monitoring window contained in $[t_s, t_f]$, then one of the random draw events performed in these windows is chosen for $E_{t_s}^{t_f}P(X)$. To avoid ambiguity, we shall give the priority to the one performed in the largest window; and if there is still a tie, then the tie will be broken by giving the priority to the one performed the latest.
- Otherwise, no event by p_i is included in $E_{t_s}^{t_f}P(X)$.

So, if $|E_{t_s}^{t_f}P(X)|=|P(X)|$, then every process in $P(X)$ has a random draw event in $E_{t_s}^{t_f}P(X)$. Furthermore, with respect to $E_{t_s}^{t_f}P(X)$, let $Q_N \subseteq P(X)$ be the set of type-N processes. For each $p_i \in Q_N$, let u_i denote the non-monitoring window in which p_i performs its random draw event chosen for $E_{t_s}^{t_f}P(X)$, and let $\|u_i\|$ denote the length of u_i. Then, the set $E_{t_s}^{t_f}P(X)$ is said to be *proper* if $t_f - t_s \leqslant \sum_{p_i \in Q_N}\|u_i\|$ and $|E_{t_s}^{t_f}P(X)|=|P(X)|$.

3.2.2. Properties of TB that hold with certainty

We now analyze the correctness of TB. We begin with the synchronization property. For this, it is useful to distinguish between an interaction (a static entity) and an instance of an interaction (a dynamic entity): when an interaction X is established, an instance of X is executed.

Theorem 1 (Synchronization). *If a process starts a new instance of X, then all other processes in $P(X)$ will eventually start the instance of X.*

[8] We say that an interval $[t_1, t_2]$ is *contained* in $[t_3, t_4]$ if $t_3 \leqslant t_1$ and $t_2 \leqslant t_4$. Two intervals *join* if they have a common end point, and they *overlap* if there exists a non-zero length interval contained in both intervals. The terms apply to semi-closed intervals as well. For example [2, 4) is contained in [1, 4], and [2, 4) and [4, 6) join.

Proof. A process starts an instance of X only if it has sent a copy of its token tagged with "X" to some $p_j \in P(X)$, and the token is returned with a tag "success". Since only the process which holds the set of tokens $\{T_j \mid p_j \in P(X), tag(T_j) = \text{"}X\text{"}\}$ can change the tags to "success", and since a process will not give up its attempt to establish X until its tokens are returned, when a process attempting X receives a token tagged with "success", all other processes in $P(X)$ will also obtain a token tagged with "success" when they retrieve their tokens. The theorem therefore follows. □

Theorem 2 (Exclusion). *No two interactions can be in execution simultaneously if they have a common member.*

Proof. This follows from the fact that a process attempts one interaction at a time. □

3.2.3. Properties of TB that hold with probability 1

We move on to prove the fairness property of TB.

Lemma 3. *Suppose that, from time $t'-u$ to time $t'+u$, for each $p_i \in P(X)$, if p_i has a non-monitoring window overlapping or joining with $[t'-u, t'+u]$, then the length of this window is strictly less than $\eta_i^{\max}$. Let $\Theta_X = \sum_{p_i \in P(X)} \eta_i^{\max}$. If X is enabled at t' and $u \geqslant \Theta_X$, then there exist t_1 and t_2, where $t' - \Theta_X < t_1 \leqslant t_2 < t' + \Theta_X$ and $t_2 - t_1 < \Theta_X$, such that $E_{t_1}^{t_2} P(X)$ is proper.*

Proof. Since X is enabled at t', each $p_i \in P(X)$ is ready for interaction at t'. So, at t', p_i is either in a non-monitoring window or in a monitoring window. It is clear that either (i) every $p_i \in P(X)$ is in a monitoring window at t', or (ii) some process in $P(X)$ is in a non-monitoring window at t'.

Consider Case (i). Let $t_1 = t_2 = t'$. By definition, then $|E_{t_1}^{t_2} P(X)| = |P(X)|$. Since with respect to $E_{t_1}^{t_2} P(X)$ there is no type-N process, set $E_{t_1}^{t_2} P(X)$ is obviously proper. Moreover, the two time instances t_1 and t_2 we have chosen easily satisfy the condition: $t' - \Theta_X < t_1 \leqslant t_2 < t' + \Theta_X$ and $t_2 - t_1 < \Theta_X$. So, the lemma is proven for this case.

Consider Case (ii). We begin with the following definition. Let U be a set of intervals $[a_j, b_j)$, where $1 \leqslant j \leqslant l$. Let $left(U) = \min\{a_j \mid 1 \leqslant j \leqslant l\}$, and $right(U) = \max\{b_j \mid 1 \leqslant j \leqslant l\}$. The intervals in U are said to be *connected* if

$$\forall t,\ left(U) \leqslant t < right(U) \Rightarrow \exists [a_k, b_k) \in U,\ a_k \leqslant t < b_k$$

(Intuitively, the intervals are connected if they can be "glued" together to form a single interval. For example, the three intervals in $\{[3,7),[5,9),[9,10)\}$ are connected, but the two intervals in $\{[3,7),[8,9)\}$ are not.) It follows from the above definition that if the intervals in U are connected, then $right(U) - left(U) \leqslant \sum_{1 \leqslant j \leqslant l} (b_j - a_j)$.

Recall that for Case (ii), there exists some process in $P(X)$, say p_1, that is in a non-monitoring window at t'. Let $[t_{1,s}, t_{1,f})$ be the non-monitoring window of p_1. Define Γ to be a set of pairs $\langle p, u \rangle$ satisfying the following conditions:

(1) For each $\langle p, u \rangle \in \Gamma$, $p \in P(X)$ and u is a non-monitoring window of p.
(2) $\langle p_1, [t_{1,s}, t_{1,f}) \rangle \in \Gamma$.

(3) For each $p \in P(X)$, Γ contains at most one pair $\langle q, u\rangle$ such that $p = q$.
(4) Let $intervals\_of(\Gamma) = \{u \mid \langle p, u\rangle \in \Gamma\}$. Then, the intervals in $intervals\_of(\Gamma)$ are connected.
(5) Γ is maximal; that is, there exists no other pair α such that set $\Gamma \cup \{\alpha\}$ satisfies the above four conditions.

(Note that there may be more than one such set.)

Let $t_1 = left(intervals\_of(\Gamma))$, and let $t_2 = right(intervals\_of(\Gamma))$. Since the intervals in $intervals\_of(\Gamma)$ are connected and since $t_{1,s} \leqslant t' < t_{1,f}$, it can be seen that $t' - \Theta_X < t_1 \leqslant t_2 < t' + \Theta_X$ and $t_2 - t_1 < \Theta_X$.

Consider $E^{t_2}_{t_1}P(X)$. Let $processes\_of(\Gamma) = \{p \mid \langle p, u\rangle \in \Gamma\}$. Clearly, with respect to $E^{t_2}_{t_1}P(X)$ each $p \in processes\_of(\Gamma)$ is a type-N process.

Let $Q = P(X) - processes\_of(\Gamma)$. We argue that, if $Q \neq \emptyset$, then with respect to $E^{t_2}_{t_1}P(X)$ each $q \in Q$ is a type-M process. To see this, observe that $t_1 \leqslant t' < t_2$ (because $t_1 \leqslant t_{1,s} \leqslant t' < t_{1,f} \leqslant t_2$). Since q does not have a non-monitoring window overlapping of joining with $[t_1, t_2)$ (for otherwise, Γ would not be maximal), q is in a monitoring window at t'. Since every monitoring window must be preceded by a non-monitoring window, and since q does not have a non-monitoring window overlappng or joining with $[t_1, t_2)$, either q remains in a monitoring window throughout $[t_1, t_2]$, or q remains in a monitoring window throughout $[t_1, t']$ and starts an interaction after the window terminates. So, with respect to $E^{t_2}_{t_1}P(X)$, q is a type-M process.

Given that, with respect to $E^{t_2}_{t_1}P(X)$, each $p \in P(X)$ is either a type-N of type-M process, we have $|E^{t_2}_{t_1}P(X)| = |P(X)|$. So to show that $E^{t_2}_{t_1}P(X)$ is proper it remains to show that $t_2 - t_1 \leqslant \sum_{p \in processes\_of(\Gamma)} \|u_p\|$, where u_p is the non-monitoring window in which p performs its random draw event chosen for $E^{t_2}_{t_1}P(X)$. For this, let v_p be the non-monitoring window of p such that $\langle p, v_p\rangle \in \Gamma$. Note that, because each $p \in processes\_of(\Gamma)$ may have more than one non-monitoring window contained in $[t_1, t_2]$, v_p and u_p may not refer to the same window. However, the u_p we have chosen to build up $E^{t_2}_{t_1}P(X)$ guarantees that $\|v_p\| \leqslant \|u_p\|$. Observe that $t_2 - t_1 \leqslant \sum_{\langle p, v_p\rangle \in \Gamma} \|v_p\|$. So, $t_2 - t_1 \leqslant \sum_{p \in processes\_of(\Gamma)} \|u_p\|$.

Therefore, the lemma is proven for Case (ii). □

Lemma 4. *Assume set $E^{t_2}_{t_1}P(X)$ is proper. With respect to $E^{t_2}_{t_1}P(X)$, let Q_N be the set of type-N processes, and Q_M be the set of type-M processes. For each $p_i \in Q_N$, let u_i denote p_i's non-monitoring window from which p_i's random draw event is chosen for $E^{t_2}_{t_1}P(X)$, and let w_i denote p_i's monitoring window immediately following u_i. For each $p_i \in Q_M$, let w_i denote p_i's monitoring window at t_1. If all the random draws in $E^{t_2}_{t_1}P(X)$ yield the same outcome X and, for each $p_i \in Q_N$, $\|w_i\| > (\sum_{p_l \in Q_N} \|u_l\|) - \|u_i\|$, then an instance of X will be started when some process $p_j \in P(X)$ finishes its monitoring window w_j.*

Proof. Since $t_2 - t_1 \leqslant \sum_{p_l \in Q_N} \|u_l\|$, and since for each $p_i \in Q_N$, p_i's monitoring window w_i has a length strictly greater than $(\sum_{p_l \in Q_N} \|u_l\|) - \|u_i\|$, p_i must still be in the monitoring window at time t_2.

Recall that every $p_j \in Q_M$ either remains in a monitoring window throughout $[t_1, t_2]$, or is monitoring an interaction at t_1 and starts the interaction after it finishes the monitoring window. Suppose first that every $p_j \in Q_M$ remains in a monitoring window throughout $[t_1, t_2]$ (where, under the lemma assumptions, this window is w_j). Then, every $p_j \in Q_M$ is also monitoring X at t_2. So, at time t_2 each process in $P(X)$ has collected every other process's token tagged with "X" and has changed (or is changing) all the tags to "success". Hence, every process $p_k \in P(X)$ will start X when it finishes its monitoring window w_k (and retrieves its tokens).

Suppose otherwise that some $p_j \in Q_M$ is monitoring an interaction at t_1 and starts the interaction after it finishes the monitoring window. Since the interaction p_j is monitoring is decided by the outcome of p_j's random draw event in $E_{t_1}^{t_2} P(X)$, by the assumptions of the lemma, the outcome is X. So, p_j will start X when it finishes its w_j.[9] □

Note that in Lemma 4 the monitoring window w_i of each $p_i \in P(X)$ must overlap or join with the interval $[t_1, t_2]$. So, if an instance of X is established and each $w_i \leqslant \delta$, then the instance will be established by time $t_2 + \delta$.

For fairness, we first show that TB satisfies weak interaction fairness, for which we need some assumption on the faultless behaviour of the system. We assume that if the communication medium remains connected, then every message will eventually reach its destination. Note that, if processes are not hanging, then they remain active (that is, every process will eventually execute its next instruction unless the instruction is a message receiving command and no message specified in the command has been sent to the process), and starting from any point the time it takes a process to execute an instruction (i.e., the length of the step to execute the instruction) will eventually be bounded.

Theorem 5 (Weak interaction fairness). *Assume that processes are not hanging and the communication medium remains connected. If X is enabled at time t then, with probability* 1, *X will be disabled eventually.*

Proof. We show that the probability is 0 that X is continuously enabled from t onward. Observe that since the communication medium remains connected and processes remain active, and since every continuously enabled guarded command will eventually be executed, a process will not be blocked indefinitely from executing its next action. So, the time it takes for each process to measure a new η value (which corresponds to the length of a non-monitoring window, although the measured value is slightly larger)

[9] In the algorithm, it is possible that some process p_1 has already started X, but another process is still monitoring X, or is even still in a non-monitoring window. For example, consider the following scenario, and assume that $P(X) = \{p_1, p_2\}$: (1) p_1 starts monitoring X; (2) p_2 randomly chooses X and sends p_1 a copy of T_2 tagged with "X"; (3) p_1 receives T_2 and acknowledges the receipt (at this point p_1 has successfully observed the establishment of X); (4) p_1 finishes its monitoring window, retrieves its tokens, and starts X; and (5) p_2 executes lines 6.1–6.2 of the algorithm, and then starts monitoring X.

is finite. Moreover, the assumption that processes are not hanging also ensures that, starting from any point, all possible η values measured by a process will eventually be bounded by some constant c. The well-founded ordering of events on the time axis ensures that a process may at most measure a finite number of distinct η values less than c.

Recall that the length of a monitoring window for p_i to monitor X is determined by the value $\sum_{p_j \in P(X)-\{p_i\}} E[j]$, where $E[j]$ is the maximum of p_j's previous η values collected between the time p_i becomes ready for interaction through the time p_i starts the monitoring window. Moreover, every time when p_i chooses to attempt X, it will learn all other participants' current η values when they acknowledge the receipt of p_i's tokens (see lines 6–6.2 of TB). Since if p_i is continuously ready it will attempt interactions infinitely often, by the *law of large numbers* (Theorem 6 will explain this law in more detail), p_i will attempt X infinitely often with probability 1. So if X is continuously enabled forever, then by the previous observations on η values, there must exist some t_0 such that, from t_0 onward, for every $p_i \in P(X)$, p_i's new η value is no greater than some $\eta_i^{\max}$, and p_i's $E[j]$ is equal to $\eta_j^{\max}$. It follows that from t_0 onward each p_i's non-monitoring window has a length less than $\eta_i^{\max}$, and each p_i's monitoring window to monitor X has a length greater than[10] or equal to $\sum_{p_j \in P(X)-\{p_i\}} \eta_j^{\max}$.

Let $\Theta_X = \sum_{p_j \in P(X)} \eta_j^{\max}$. Consider the interval $[t_0, t_0 + 2\Theta_X)$. Given that from t_0 onward each p_i's non-monitoring window has a length less than $\eta_i^{\max}$, Lemma 3 (with $t' = t_0 + \Theta_X$ and $u = \Theta_X$) ensures that there exist two time instances $t_{1,s}, t_{1,f}$, where $t_0 < t_{1,s} \leqslant t_{1,f} < t_0 + 2\Theta_X$ such that $E_{t_{1,s}}^{t_{1,f}} P(X)$ is a proper set of random draw events. Given that starting from t_0 each p_i's non-monitoring window has a length less than $\eta_i^{\max}$, and each p_i's monitoring window to monitor X has a length greater than or equal to $\sum_{p_j \in P(X)-\{p_i\}} \eta_j^{\max}$, Lemma 4 implies that, if the random draws in $E_{t_{1,s}}^{t_{1,f}} P(X)$ yield the same outcome X, then X will be disabled. Note that, even if the random draws do not yield the same outcome, some process in $P(X)$ may still establish another interaction X' if its random draw coincides with other processes' random draws.

Let μ denote the probability that X remains enabled starting from t up to the point the random draws in $E_{t_{1,s}}^{t_{1,f}} P(X)$ are to be made. So the probability that the random draws in $E_{t_{1,s}}^{t_{1,f}} P(X)$ do not cause X to be disabled is no greater than $\mu(1-\psi_X)$, where ψ_X is the probability that the random draws in $E_{t_{1,s}}^{t_{1,f}} P(X)$ yield the same outcome X. If X remains enabled after the random draws, then every process in $P(X)$ will perform a new random draw in finite time, and so by Lemma 3 again there exists another proper set of random draws $E_{t_{2,s}}^{t_{2,f}} P(X)$ such that $E_{t_{1,s}}^{t_{1,f}} P(X) \cap E_{t_{2,s}}^{t_{2,f}} P(X) = \emptyset$. The probability that X remains enabled after the new set of random draws is no greater than $\mu(1-\psi_X)^2$. In

[10] The length may be greater than $\sum_{p_j \in P(X)-\{p_i\}} E[j]$ because the condition that the length of p_i's monitoring window equals to $\sum_{p_j \in P(X)-\{p_i\}} E[j]$ only causes the guarded command in line 15′ to be enabled; it is not necessarily executed right away.

general, the probability that X remains enabled after l mutually disjoint sets of random draws is no greater than $\mu(1-\psi_X)^l$. If X continues to be enabled then l will keep increasing and, so, $\mu(1-\psi_X)^l$ tends to 0. So the probability that X remains enabled forever is 0. □

Theorem 6 (Strong interaction fairness). *Assume* (A1) *that processes are not hanging and the communication medium remains connected, and* (A2) *that a process's transition to a state ready for interaction does not depend on the random draws performed by other processes. If an interaction X is enabled infinitely often then, with probability* 1, *the interaction will be executed infinitely often.*

Proof. Assume the hypothesis that X is enabled infinitely often. By (A1), there exists some time instance t_0 after which every non-monitoring window of p_k has a length less than $\eta_k^{\max}$ for each p_k in the system, and every monitoring window of p_k has a length no less than $\Theta_X - \eta_k^{\max}$, where $\Theta_X = \sum_{p_j \in P(X)} \eta_j^{\max}$. Because t_0 is finite, from t_0 onward X is still enabled infinitely often. By Lemma 3, there exist infinitely many t_i's, $t_{i,1}$'s, and $t_{i,2}$'s, where $i>0$, $t_i - \Theta_X < t_{i,1} \leqslant t_{i,2} < t_i + \Theta_X$ and $t_{i,2} - t_{i,1} < \Theta_X$, such that X is enabled at $t_i, E_{t_{i,1}}^{t_{i,2}} P(X)$ is proper, and $E_{t_{i,1}}^{t_{i,2}} P(X) \cap E_{t_{j,1}}^{t_{j,2}} P(X) = \emptyset$ if $i \neq j$. Let $\mathbb{I}$ be the set of indices of such t_i's. By Lemma 4, if the random draws in $E_{t_{i,1}}^{t_{i,2}} P(X)$ yield the same outcome X, then an instance of X will be established. So, in the following, we shall show that the probability is 1 that there are infinitely many i's in $\mathbb{I}$ such that $E_{t_{i,1}}^{t_{i,2}} P(X)$ yield the same outcome X. This then establishes the theorem.

Because $\mathbb{I}$ is infinite and there are only a finite number of interactions in the system, there exists an infinite subset $\mathbb{J} \subset \mathbb{I}$ such that, for each $p \in P(X)$, p is ready for the same set of interactions $\mathbb{A}_p$ at t_i for each $i \in \mathbb{J}$. Let $\psi_{\mathbb{A}_p,X}$ be the non-zero probability that X is chosen from $\mathbb{A}_p$ in a random draw. Let $\psi_X = \prod_{p \in P(X)} \psi_{\mathbb{A}_p,X}$. Consider $E_{t_{i,1}}^{t_{i,2}} P(X)$, where $i \in \mathbb{J}$. By Assumption (A2), the random draws in $E_{t_{i,1}}^{t_{i,2}} P(X)$ are independent of the enabledness of X at t_i and, so, are independent of one another. So, the probability that the random draws in $E_{t_{i,1}}^{t_{i,2}} P(X)$ produce the same outcome X is ψ_X.

For each $i \in \mathbb{J}$, define random variable E_i to be 1 if the random draws in $E_{t_{i,1}}^{t_{i,2}} P(X)$ produce the same outcome X, and 0 otherwise. Then $E_i = 1$ also has the probability ψ_X. Let the indices of $\mathbb{J}$ be enumerated by $j_1, j_2, \ldots$. By the *law of large numbers* in probability theory (see, for example, the book by Chung[9]), for any given ε we have

$$\lim_{n \to \infty} \boldsymbol{P}\left(\left| \frac{\sum_{1 \leqslant i \leqslant n} E_{j_i}}{n} - \psi_X \right| \leqslant \varepsilon \right) = 1.$$

That is, when n tends to infinity, the probability is 1 that $(\sum_{1 \leqslant i \leqslant n} E_{j_i})/n$ tends to ψ_X. Therefore, with probability 1, the set $\{i \mid E_{j_i} = 1,\ i \geqslant 1\}$ is infinite. So, with probability 1, there are infinitely many i's in $\mathbb{J}$ such that the random draws in $E_{t_{i,1}}^{t_{i,2}} P(X)$ yield the same outcome X. Hence, with probability 1 there are infinitely many i's in

$\mathbb{I}$ such that the random draws in $E_{t_{i,1}}^{t_{i,2}} P(X)$ yield the same outcome X. The theorem is therefore proven.[11] □

Like the algorithm presented in [19], a conspiracy against strong interaction fairness can be devised if Assumption (A2) is dropped from Theorem 6. To see this, consider a system of two processes p_1 and p_2, and three interactions X_1, X_2, and X_{12}, where $P(X_1) = \{p_1\}$, $P(X_2) = \{p_2\}$, and $P(X_{12}) = \{p_1, p_2\}$. Assume that p_1 is ready for both X_1 and X_{12}. So it will toss a coin to choose one to attempt. The malicious p_2 could stay in its local computing phase until p_1 has randomly selected X_1; then $p2$ becomes ready for X_2 and X_{12} before p_1 executes X_1. Since p_1's attempt to execute X_1 will succeed once it selects X_1, X_{12} will not be executed this time. However, X_{12} is enabled as soon as p_2 becomes ready. Similarly, p_1 could also stay in its local computing phase until p_2's random draw yields X_2. So if this scenario is repeated over and over again, then the resulting computations would not be strong interaction fair. Note that in the resulting computation there exist infinite many $t_{i,1}$'s and $t_{i,2}$'s such that $E_{t_{i,1}}^{t_{i,2}} P(X_{12})$ is proper. However, the two random draws in $E_{t_{i,1}}^{t_{i,2}} P(X_{12})$ are *not* mutually dependent because one of them is performed only if the other has outcome X_1 (or X_2)

3.2.4. Time complexity

To measure the time complexity of TB, we wish to know that, when an interaction X is enabled, how long it takes a participant of X to execute an interaction, i.e., to disable X.[12] It can be seen from Theorem 5 that a necessary condition for X to be disabled is that processes' speeds will not keep decreasing. So, to simplify the analysis, we shall first consider a stable system where processes' speeds do not vary. Moreover, for subsequent comparison with deterministic algorithms, we shall also simplify the

[11] The *law of large numbers* cannot be used to prove the theorem if one were to reset time variables $E[j]$ periodically. This is because although there are infinitely many i's in $\mathbb{I}$ such that all the random draws in each $E_{t_{i,1}}^{t_{i,2}} P(X)$ yield the same outcome X, Lemma 4 might not be used to guarantee the establishment of X because each process's monitoring window following its random draw in $E_{t_{i,1}}^{t_{i,2}} P(X)$ could incidentally be reset to a value unable to satisfy the condition of Lemma 4. Instead, the second Borel–Cantelli Lemma can be used to prove the theorem. As a consequence of the lemma, it is a well known fact in measure theory and probability that (see for instance Example 4.14 of [6]), if a coin (with outcome 0 or 1) is tossed an infinite number of times, then given any constant c the probability is 1 that there are infinitely many runs of 1 of length greater than c (where a *run* of 1 is a sequence of 1's surrounded by two 0's; its length is the number of 1's in the sequence).

Given that the length of a non-monitoring window will eventually be bounded, we can see that, from some point onward, if an interaction X is enabled and each participant of X always chooses X to attempt, then after at most some finite number of attempts X will be established (they failed to establish X in earlier attempts because their monitoring windows were too short to satisfy the condition of Lemma 4). The above fact in measure theory and probability guarantees that, if X is enabled infinitely often, then the probability is 1 that, infinitely often, every participant of X will continuously choose X to attempt for at least some finite number of times. Therefore, the probability is 1 that X will be established infinitely often.

[12] Given that interactions' membership rosters may overlap, it is clear that no algorithm can guarantee the following: when an interaction is enabled, then this particular instance of interaction must eventually be executed with certainty; for, otherwise, the exclusion requirement of the interaction scheduling would not be satisfied.

analysis by assuming that each non-monitoring window takes a constant time $\eta - \varepsilon$ for some $\varepsilon > 0$, and each interaction involves m participants. By the algorithm, each monitoring window then must take more than $(m-1)(n-\varepsilon)$ time. Let us assume that it takes $(m-1)\eta + \varepsilon$ time.

Theorem 7 (Time complexity). *Suppose each interaction involves m participants. Suppose further that each non-monitoring window has a length $\eta - \varepsilon$ for some $\varepsilon > 0$, and each monitoring window has a length $(m-1)\eta + \varepsilon$. Then, once an interaction X is enabled, the expected time it takes for a member of X to start an interaction is no greater than*

$$\frac{m\eta}{\prod_{p_i \in P(X)} \psi_{p_i,X}} + (m-1)\eta + \varepsilon$$

where $\psi_{p_i,X}$ is the probability that p_i chooses X in its random draw.

Proof. Assume the hypothesis, and that X is enabled at time t. By Lemma 3 (with $\eta_i^{\max} = \eta$, $t' = t$, $u = \Theta_X = m\eta$), there exist two time instances t_1 and t_2, where $t - m\eta < t_1 \leqslant t_2 < t + m\eta$ and $t_2 - t_1 < m\eta$, such that $E_{t_1}^{t_2} P(X)$ is proper. By Lemma 4 (with the hypothesis that each monitoring window has a length $(m-1)\eta + \varepsilon$ satisfying the condition: $(m-1)\eta + \varepsilon > (m-1)(\eta - \varepsilon)$) and the remark following the lemma, if the random draws in $E_{t_1}^{t_2} P(X)$ yield the same outcome X (an event that occurs with probability $\psi_X = \prod_{p_i \in P(X)} \psi_{p_i,X}$), then an instance of X will be established by time $t_2 + (m-1)\eta + \varepsilon < t + m\eta + (m-1)\eta + \varepsilon$. Note that if the random draws do not yield the same outcome X but some process's random draw in $E_{t_1}^{t_2} P(X)$ leads to the establishment of some other interaction involving the process, then the process will also start an interaction when it finishes its monitoring window (that is established following the random draw). If neither of these is the case then each process in $P(X)$, after performing its random draw in $E_{t_1}^{t_2} P(X)$, must perform a new random draw in another $m\eta$ time (which amounts to the length of a non-monitoring window $\eta - \varepsilon$ plus the length of a monitoring window $(m-1)\eta + \varepsilon$). That is, there must exist another proper set of random draws $E_{t_1+m\eta}^{t_2+m\eta} P(X)$ that is disjoint from $E_{t_1}^{t_2} P(X)$.

Once again, if the new random draws yield the same outcome X or cause some other interaction to be established (with probability no less than $(1-\psi_X)\psi_X$), then some interaction involving a member of X will be established by time $t_2 + m\eta + (m-1)\eta + \varepsilon < t + 2m\eta + (m-1)\eta + \varepsilon$. Otherwise, there must exist another proper set of random draws $E_{t_1+2m\eta}^{t_2+2m\eta} P(X)$ that is disjoint form $E_{t_1+m\eta}^{t_2+m\eta} P(X)$, and so on.

In general, if X remains enabled, then there exist mutually disjoint sets of random draws $E_{t_1}^{t_2} P(X)$, $E_{t_1+m\eta}^{t_2+m\eta} P(X), \ldots$, $E_{t_1+(i-1)m\eta}^{t_2+(i-1)m\eta} P(X), \ldots$, and each of these sets is proper. Moreover, if the random draws in $E_{t_1+(i-1)m\eta}^{t_2+(i-1)m\eta} P(X)$ yield the same outcome X or cause some other interaction to be established (with probability no less than $(1-\psi_X)^{i-1}\psi_X$), then an interaction involving a member of X will be established by $t + im\eta + (m-1)\eta + \varepsilon$. Therefore, the expected time starting from t until an interaction involving a member

of X is established is less than

$$\left(\sum_i im\eta(1-\psi_X)^{i-1}\psi_X\right)+(m-1)\eta+\varepsilon=\frac{m\eta}{\psi_X}+(m-1)\eta+\varepsilon. \qquad \square$$

Similar analysis can also be carried out if interactions have different size or non-monitoring windows have different lengths. In particular, when the length of p_j's non-monitoring windows may vary, another process p_i must update its $E[j]$ in order to adjust its monitoring window for monitoring some interaction involving p_j. In the algorithm, p_i learns a new η_j (which measures the maximum length of p_j's previous non-monitoring windows) through an attempt to establish an interaction involving p_j. For p_i to have such an attempt it must choose an interaction involving p_j in some random draw. Let $\mu_{i,j}$ denote the probability that, in one random draw by p_i, an interaction involving p_j is chosen. Then the expected number of attempts for p_i to finally attempt an interaction involving p_j so as to update p_i's $E[j]$ is

$$T_{i,j}=\sum_k k(1-\mu_{i,j})^{k-1}\mu_{i,j}=\frac{1}{\mu_{i,j}}.$$

If each such attempt takes no more than s time (which consists of a non-monitoring window followed by a monitoring window), then an additional $s/\mu_{i,j}$ time would be required for p_i to have the knowledge of p_j's new η_j. If p_j also has no knowledge of p_i's new η_i, then an additional $\max\{T_{i,j},T_{j,i}\}\cdot s$ time would be required for both p_i and p_j to have each other's new η.

To see how the time complexity is affected by (1) the number of potential interactions for which a process may be ready at a time, and (2) the size of an interaction, assume that a process may be ready for k potential interactions at a time, and each interaction involves m participants. So the probability for the processes in $P(X)$ to choose X in a set of random draws, one by each process, is $(1/k)^m$. Assume further that each non-monitoring window has a length $\eta-\varepsilon$ and a monitoring window has a length $(m-1)\eta+\varepsilon$. From Theorem 7, the expected time for an enabled interaction to be disabled is dominated by $mk^m\eta$. Suppose that the time to execute a local action is negligible compared to the communication time for delivering a message. Then, η consists of four message transmissions (a message to send the token, an acknowledgement, a message to retrieve the token, and a message to return the token) if messages in lines 5, 6, 16, and 17 of TB can be sent in parallel. If the message transmission time is c, then the time complexity is dominated by

$$4cmk^m.$$

In the above, since m messages are sent in parallel in each interval c, the expected number of messages needed to establish an interaction per process is no greater than

$$4m^2k^m.$$

For comparison, the efficient deterministic alogrithm by Ramesh [28] has a worst case time complexity in the order of $3cnk$ and a message complexity $3mk$. Note that, unlike

TB (and other randomized algorithms [12, 30, 19]), the time complexity of deterministic algorithms typically depends on n — the total number of processes in the system. This is because they impose priority (e.g., process id's) to break the symmetry between processes so that a low-priority process must wait for a high-priority one if they attempt to establish conflicting interactions (two interactions *conflict* if they involve a common process).[13] The fact that randomized algorithms often have a time complexity independent of n is one of the reasons that Reif and Spirakis's randomized algorithm [30] was able to claim a real-time response.

From the above comparison, we can see that TB can out-perform deterministic algorithms (where only WIF is required) only if time is a main concern and the two parameters, k — the number of potential interactions for which a process may be ready at a time, and m — the number of participants in an interaction, are kept small relative to n, e.g., CSP-like biparty interactions. (For efficiency's concern, deterministic or randomized, it is generally known that the two parameters must be kept small in practical applications. A technique of *synchrony loosening* [10] is therefore proposed for reducing the size of an interaction.) Otherwise, TB has a niche simply because deterministic algorithms are unable to guarantee SIF.

4. A shared-memory solution

In this section we present an algorithm for the multiparty interaction scheduling problem where processes communicate by reading from and writing to shared variables. In particular, the algorithm uses only single-writer variables. A non-local variable V_j can be read by the command *read* (V_j).

4.1. Informal description

Like Algorithm TB, when a process p_i is ready for interaction, it randomly chooses one interaction X, from the set of potential interactions it is ready to execute, and then attempts to establish X. However, instead of sending out tokens, p_i expresses its interest in X by writing $\langle examining, X\rangle$ to its local variable *state*, which is to be read by other processes. In the algorithm, values of *state* is of the form $\langle status, X\rangle$, where X denotes the interaction p_i is attempting, and *status* records the status of the attempt. Besides *examining*, *status* has another three possible values: *waiting*, *success*, and *closed*; their meaning should be clear shortly.

After setting its state to $\langle examining, X\rangle$, p_i begins to read the states of the other participants. If, for every $p_j \in P(X)$, p_j's *state* is $\langle examining, X\rangle$ or $\langle waiting, X\rangle$, then the other processes in $P(X)$ are also interested in X. This means that p_i has successfully observed the establishment of X. It then changes its state to $\langle success, X\rangle$, and waits for the other participants to observe the establishment of X. To do so, p_i keeps a

[13] It is well known that, even if only WIF is required, there is still no symmetric, decentralized, and deterministic algorithm for scheduling process interactions [12, 21].

binary variable $flag[X]$ for each interaction X. Initially, all processes in $P(X)$ have their $flag[X]$'s set to the same value, say 0. When a process p is to execute an instance of X, it complements its $flag[X]$. In the above case, p_i complements its $flag[X]$ before it changes its state to $\langle success, X\rangle$. To ensure that every other $p_j \in P(X)$ has also observed the establishment of X, p_i keeps reading p_j's $flag[X]$ until it has the same value as p_i's. Then, p_i changes its state to $\langle closed, X\rangle$ and starts X.

As we shall see, $flag[X]$ has another important role in the algorithm: to avoid a process from "outrunning" other processes in executing instances of X. In other words, the algorithm guarantees that, if p_i is to execute an instance of X, then all other processes in $P(X)$ must have finished the previous instance of X.

When examining other processes' states, if not all of them are $\langle examining, X\rangle$ or $\langle waiting, X\rangle$, then p_i changes its state to $\langle waiting, X\rangle$. Like TB, p_i has to wait for a period of time Δ, and then re-inspects the other participants' states. The value of Δ is determined as in TB. That is, Δ must be no less than $\sum_{p_j \in P(X)-\{p_i\}} \eta_j$, where η_j is the maximum time (measured by the algorithm) p_j has spent between two consecutive Δ-intervals.

If after Δ time some process p_j has changed its state to $\langle success, X\rangle$, and $p_j.flag[X] \neq p_i.flag[X]$, then p_i has learned the establishment of X from p_j. (Throughout the paper we often use $p_j.v$ to denote p_j's variable v.) So, p_i also complements its $flag[X]$ and then starts X. If after Δ time either (1) no process's state has changed to $\langle success, X\rangle$, or (2) some process is in state $\langle success, X\rangle$ but its $flag[X]$ has the same value as $p_i.flag[X]$ (which means that the process is still executing the previous instance of X), then p_i's attempt to establish X has failed. It must return to the beginning of the procedure to attempt another interaction.

4.2. The code

The algorithm executed by each process p_i is given in Fig. 5. We shall refer to the algorithm as SM (for Shared Memory). The variables local to p_i are given as follows:

- *ready*: a boolean flag that is set to true when p_i is ready for interaction, and is set to false when p_i has executed some interaction. It is initialized to false.
- $state[1..n]$: array of $\langle status, X\rangle$, where X is an interaction, and *status* is *examining*, *waiting*, *success*, or *closed*. Each $state[j]$ records the state of p_j observed by p_i, and is initialized to $\langle closed, \perp\rangle$.
- $flag[X_1..X_m]$: array of binary values, where $X_1, \ldots, X_m$ are interactions of which p_i is a member. Each $flag[X_j]$ is initialized to 0.
- η: η records the maximum of the durations from the time p_i previously stopped monitoring interaction to the time p_i starts monitoring interaction. It is initialized to 0.
- $init\_\eta$: a temporary variable used to measure η. It is initialized to ∞.
- $E[1..n]$: $E[j]$, initialized to 0, records the maximum value of p_j's η read by p_i.

In the algorithm, variable η is measured in a way similar to TB. That is, p_i starts timing η before it is ready for interaction (line 3), and before it is to stop monitoring

```
1   while true do {
2       local computing; /* in local computing phase */
3       init_η := clock(p_i); /* start measuring a new η */
4       ready := true;
5       while ready do { /* ready for interaction */
6           randomly select an interaction X for which p_i is ready to execute;
7           state[i] := ⟨examining, X⟩;
8           for p_j ∈ P(X), p_j ≠ p_i do {
9               state[j] := read(p_j.state[j]);
10              E[j] := max(read(p_j.η), E[j]); }
            /* start monitoring X */
11          η := max(η, clock(p_i) − init_η); /* record a new η */
12          if ∀p_j ∈ P(X) : state[j] ∈ {⟨examining, X⟩, ⟨waiting, X⟩} then {
13            /* p_i has successfully observed the establishment of X */
14            flag[X] := ¬flag[X];
15            state[i] := ⟨success, X⟩;
16            for p_j ∈ P(X), j ≠ i do
                  /* wait for p_j to learn the establishment of X */
17                while read(p_j.flag[X]) ≠ flag[X] do;
18            state[i] := ⟨closed, X⟩;
              /* stop monitoring X */
19            execute X;
20            ready := false; }
21          else { /* p_i is unable to observe the establishment of X */
22              state[i] := ⟨waiting, X⟩;
23              wait for Δ = Σ_{p_j ∈ P(X)−{p_i}} E[j] time;
24              init_η := clock(p_i); /* start measuring a new η */
25              state[i] := ⟨closed, X⟩;
                /* stop monitoring X */
26              for p_j ∈ P(X), j ≠ i do { /* re-inspect other process's state */
27                state[j] := read(p_j.state[j]);
28                while state[j] = ⟨examining, X⟩ do
29                  state[j] := read(p_j.state[j]);
30                if state[j] = ⟨success, X⟩ and read(p_j.flag[X]) ≠ flag[X]
31                  then { /* p_j has observed the establishment of X; it
32                            then executes X and returns to an idle state. */
33                      flag[X] := ¬flag[X];
34                      execute X;
35                      ready := false;
36                      break; /* exit the for-loop */ }
37              } /* end of for-loop */
38          } /* end of else statement */
39      } /* end of while-loop */
40  } /* end of while-loop */
```

Fig. 5. Algorithm SM.

an interaction (line 24). A new η value is recorded in line 11 when p_i is to wait for another Δ-interval (i.e., to start monitoring some interaction). The value is to be read by other processes (line 10) for them to adjust their Δ-intervals (line 23).

It is important to note that when a process p_i has observed the establishment of a new instance of X, it must complement its $flag[X]$ *before* changing its state to $\langle success, X\rangle$ (lines 14–15). Otherwise, some process p_j, after observing p_i's state $\langle success, X\rangle$ (lines 27–29), could have read the value of $p_i.flag[X]$ before the complement and then regard p_i as still in a previous instance of X. So, p_j would not commit to X albeit p_i has already committed, thus violating the synchronization property of the problem. (The crucial role of this ordering can be seen in the proof of Lemma 8.)

Furthermore, when p_i is re-inspecting p_j's state in lines 28–29, if p_j is in state $\langle examining, X\rangle$, then p_i must wait until p_j leaves the examining status. This is because p_i cannot be sure whether p_j will then enter state $\langle success, X\rangle$ or $\langle waiting, X\rangle$. In the former case p_i may start an instance of X, while in the latter p_i should return to the beginning of the algorithm to attempt another interaction. Note that, there is no danger of deadlock because p_j in state $\langle examining, X\rangle$ will not be blocked by p_i (or any other process).

4.3. *Analysis of* SM

We now analyze the correctness of SM. We begin with an invariant of the algorithm.

Lemma 8. *At any time of the algorithm either* (1) *all the* $p_j.flag[X]$*'s, where* $p_j \in P(X)$, *have the same value, or* (2) *if the* $p_j.flag[X]$*'s are different, then there exists some previous time instance* t *such that all the* $p_j.flag[X]$*'s were equal at time* t, *and there exists another time instance* t' *such that all the* $p_j.flag[X]$*'s will be equal at* t' *and, in between* t *and* t', *every* p_j *complements its* $flag[X]$ *only once.*

Proof. Let $t_1, t_2, \ldots$ be the time instances on the global time axis where the events of the system are totally ordered, and let t_0 be the initial time. We shall prove a stronger invariant $\mathbb{INV}$ that not only guarantees the condition described in the lemma (henceforth referred to as $\mathbb{INV}_1$), but also ensures the following condition $\mathbb{INV}_2$: if all the $p_j.flag[X]$'s (where $p_j \in P(X)$) are equal, then the state of the system guarantees that the next event that can make these $p_j.flag[X]$'s different must be the execution of the complement statement in line 14.

It is easy to see that $\mathbb{INV}_1$ holds at t_0 because all the $p_j.flag[X]$'s are initialized to the same value. For $\mathbb{INV}_2$, we note that a process can change its $flag[X]$ only if (a) it is in state $\langle examining, X\rangle$ and has observed the establishment of X—i.e., has observed that every other process in $P(X)$ is in state $\langle examining, X\rangle$ or $\langle waiting, X\rangle$ (line 14), or (b) it is in state $\langle closed, X\rangle$ and while re-inspecting the other processes' states, it finds that some process in $P(X)$ has already reached state $\langle success, X\rangle$ and their $flag[X]$'s are different (line 33). Given that each process's state is initialized to $\langle closed, \perp\rangle$, and that all the $p_j.flag[X]$s are initialized to the same

value, it is easy to see that no process in $P(X)$ can later change its $flag[X]$ via the complement statement in line 33 without some other process in $P(X)$ to first change its $flag[X]$ via the complement statement in line 14. Therefore, both $\mathbb{INV}_1$ and $\mathbb{INV}_2$ hold at t_0.

For the induction proof, we shall assume that $\mathbb{INV}$ holds at t_{l-1}, $l>0$. Moreover, all the $p_j.flag[X]$'s have the same value (say 0) at t_{l-1}, but some process p_i changes its $flag[X]$ at t_l to cause the $p_j.flag[X]$'s to be different at t_l. We shall show that there exists some time $t_{l'}$ such that all the $p_j.flag[X]$'s will become equal (with value 1) at $t_{l'}$, and $\mathbb{INV}$ holds throughout $[t_l, t_{l'}]$.

By the induction hypothesis, p_i must change its $flag[X]$ at t_l via the complement statement in line 14. So, p_i has observed the establishment of X prior to t_l. Recall the algorithm that after p_i has complemented its $flag[X]$ to 1, it changes its state to $\langle success, X\rangle$, and executes the for-loop in lines 16–17 until $p_j.flag[X]$ is changed to 1 for every other $p_j \in P(X)$. Consider each such p_j, and recall that $p_j.state \in \{\langle examining, X\rangle, \langle waiting, X\rangle\}$ when p_i inspected it in line 9. Since p_i will not exit the for-loop of lines 16–17 until $p_j.flag[X]$ is set to 1, to show that $t_{l'}$ exists, we first show that p_j will eventually set its $flag[X]$ to 1.

Suppose first that p_j was in state $\langle examining, X\rangle$ when p_i inspected its state. By the algorithm, p_j will eventually enter $\langle success, X\rangle$ or $\langle waiting, X\rangle$, depending on if p_j can also observe the establishment of X. If p_j can also observe the establishment of X then, like p_i, p_j enters state $\langle success, X\rangle$, complements its $flag[X]$ to 1, and will also be waiting in lines 16–17 until all other processes in $P(X)$ have the same value of $flag[X]$'s. The case that p_j instead enters state $\langle waiting, X\rangle$ is collaterally considered in the following where some process was in state $\langle waiting, X\rangle$ when inspected by p_i.

Suppose instead that p_j was in state $\langle waiting, X\rangle$ when p_i inspected its state. Then, p_j must be in lines 23–24 when p_i inspected its state. So, after p_j's Δ expires p_j must re-inspect other processes' states. Observe that p_i changed its state to $\langle examining, X\rangle$ before it inspected p_j's state. So, when p_j re-inspects p_i's state, either p_i is still in state $\langle examining, X\rangle$ inspecting other processes' states, or it has already finished the inspection and has changed its state to $\langle success, X\rangle$, waiting in lines 16–17 for p_j (and every other process in $P(X)$) to complement its $flag[X]$. Since p_j cannot finish re-inspecting p_i's state until p_i has left state $\langle examining, X\rangle$, p_j will eventually learn that p_i's state is $\langle success, X\rangle$. Moreover, since p_i complements $p_i.flag[X]$ before changing its state to $\langle success, X\rangle$, and since p_j inspects $p_i.flag[X]$ after it sees that p_i is in state $\langle success, X\rangle$, when p_j inspects $p_i.flag[X]$, it must learn that $p_i.flag[X] \neq p_j.flag[X]$ and so will set $p_j.flag[X]$ to 1.

So, we see that every process in $P(X)$ will eventually set its $flag[X]$ to 1. To complete the proof that $t_{l'}$ exists, we must show that before these $flag[X]$'s are set to 1, each process can only complement its $flag[X]$ once (starting from t_l). Note that, if some process has not yet complemented its $flag[X]$ to 1, then p_i (and all other processes that have observed the establishment of X) must stay in the for-loop in lines 16–17. So it suffices to show that, for each $p_j \in P(X)$ that does not observe the establishment of X by itself, p_j cannot reset its $flag[X]$ to 0 while p_i (or any other

process that has observed the establishment of X) is still in the for-loop. For this, observe that for p_j to reset its $flag[X]$ to 0, p_j must re-enter state $\langle examining, X\rangle$ in line 7. So, when p_j inspects the other participants' states in line 9, it will find that p_i is still in state $\langle success, X\rangle$ and so cannot proceed to line 14 to reset its $flag[X]$. Moreover, when p_j subsequently enters state $\langle waiting, X\rangle$ and re-inspects p_i's state in lines 27–29, if p_i is still in the for-loop, then when p_j proceeds to line 30, p_j will learn that $p_j.flag[X] = p_i.flag[X]$ and so will not be able to reset $p_j.flag[X]$ to 0.

Therefore, there exists $t_{l'}$ such that all the $p_j.flag[X]$'s (where $p_j \in P(X)$) will become equal at $t_{l'}$. The fact that each p_j can only complement its $flag[X]$ once throughout $[t_l, t_{l'}]$ and the assumption that all the $p_j.flag[X]$'s are equal at t_{l-1} imply that $\mathbb{INV}_1$ holds throughout $[t_l, t_{l'}]$.

We now show that $\mathbb{INV}_2$ holds throughout $[t_l, t_{l'}]$. Because $\mathbb{INV}_2$ holds vacuously if the $p_j.flag[X]$'s are different, it suffices to show that the system state at $t_{l'}$ guarantees that the next event to reset any of these $p_j.flag[X]$'s to 0 must be the complement statement in line 14. For this, in the above proof we have seen that, while some process p_i is in state $\langle success, X\rangle$ waiting for all processes in $P(X)$ to set their $flag[X]$'s to 1, no other process p_j in $P(X)$ can proceed to line 33 to complement $p_j.flag[X]$ to 0. Therefore, if after $t_{l'}$ some process $p_k \in P(X)$ has observed that another process p_h is in state $\langle success, X\rangle$ and their $flag[X]$'s are different (so that p_k can reset its $flag[X]$ to 0 via the complement statement in line 33), then the fact that p_h can be in state $\langle success, X\rangle$ must be due to the fact that p_h has reset its $flag[X]$ to 0 via the complement statement in line 14 at some time after $t_{l'}$ (but before p_k has reset its $flag[X]$ to 0). So, the first event after $t_{l'}$ to reset any $flag[X]$ to 0 must be the complement statement in line 14.

Therefore, both $\mathbb{INV}_1$ and $\mathbb{INV}_2$ hold throughout $[t_l, t_{l'}]$. The lemma is thus proven. □

The following lemma follows immediately from the above proof.

Lemma 9. *A process entering state $\langle success, X\rangle$ of SM will eventually execute an instance of X.*

The synchronization property of SM follows from Lemma 8 and the fact that every complement of $flag[X]$ is followed by an execution of X.

Theorem 10 (Synchronization). *If a process starts a new instance of X, then all other processes in $P(X)$ will eventually start the instance of X.*

The exclusion property follows directly from the fact that a process attempts interactions one at a time.

Theorem 11 (Exclusion). *No two interactions can be in execution simultaneously if they have a common member.*

To show that SM satisfies weak and strong interaction fairness, again we need some definitions about monitoring windows, non-monitoring windows and proper sets of random draws $E_{t_s}^{t_f}P(X)$. Analogous to the analysis of TB, we say that p_i *starts monitoring* X if it has set its state to $\langle examining, X\rangle$ and has finished reading the state of every other process in $P(X)$ (lines 8–10 of SM). It *stops monitoring* X if it has changed its state to $\langle closed, X\rangle$ (lines 18 or 25). Let t_1 and t_2 denote the two events respectively. The semi-closed interval $[t_1, t_2)$ is referred to as a *monitoring window*, and at any time instance of the interval p_i is monitoring X. Note that if p_i is monitoring X, then it must be in state $\langle examining, X\rangle$, $\langle success, X\rangle$, or $\langle waiting, X\rangle$. Accordingly, the definitions of *non-monitoring window*s and $E_{t_s}^{t_f}P(X)$ can be defined as in Section 3.2.1.

Like TB, the definition of "monitoring windows" ensures that if every process in $P(X)$ is monitoring X, then X will be established, as shown in the following lemma.

Lemma 12. *Assume set $E_{t_1}^{t_2}P(X)$ is proper. With respect to $E_{t_1}^{t_2}P(X)$, let Q_N be the set of type-N processes, and Q_M be the set of type-M processes. For each $p_i \in Q_N$, let u_i denote p_i's non-monitoring window from which p_i's random draw event is chosen for $E_{t_1}^{t_2}P(X)$, and let w_i denote p_i's monitoring window immediately following u_i. For each $p_i \in Q_M$, let w_i denote p_i's monitoring window at t_1. If all the random draws in $E_{t_1}^{t_2}P(X)$ yield the same outcome X and, for each $p_i \in Q_N$, $\|w_i\| > (\sum_{p_l \in Q_N} \|u_l\|) - \|u_i\|$, then an instance of X will be started when some process $p_j \in P(X)$ finishes its monitoring window w_j.*

Proof. By a proof similar to Lemma 4, we can show that all processes in Q_N are monitoring X at time t_2. Moreover, every $p_j \in Q_M$ either remains in a monitoring window throughout $[t_1, t_2]$, or is monitoring an interaction at t_1 and starts the interaction after it finishes the monitoring window. In the first case, we can see that all the processes in $P(X)$ are monitoring X at t_2; and, in the later case, it is easy to see that X will be established when some process $p_j \in Q_M$ finishes its monitoring window w_j. So, in the following we shall only show that if all processes are monitoring X at t_2, then an instance of X will be established when they finish their monitoring windows.

By definition, each process must be in state $\langle examining, X\rangle$, $\langle success, X\rangle$, or $\langle waiting, X\rangle$ at time t_2. So it suffices to consider the following two cases: (1) Some process p_i is in state $\langle success, X\rangle$ executing the for-loop in lines 16–17 of the algorithm, or (2) all processes are in state $\langle examining, X\rangle$ or $\langle waiting, X\rangle$.

For Case (1), by Lemma 9 and Theorem 10, the processes in $P(X)$ will start an instance of X when their monitoring windows at t_2 expire.

For Case (2), observe that a process can enter state $\langle waiting, X\rangle$ only from state $\langle examining, X\rangle$. Let p_l be the process, among the processes in $P(X)$, that is the last to enter state $\langle examining, X\rangle$ (i.e., to execute line 7), and assume that p_l entered the state at t', where $t' < t_2$. (If there is more than one such process, then choose an arbitrary one.) Moreover, since p_l is monitoring X at t_2, p_l, after entering state $\langle examining, X\rangle$ at t', must have finished inspecting the other participants' states by

t_2. Since every $p_i \in P(X)$ is in state $\langle examining, X\rangle$ or $\langle waiting, X\rangle$ throughout the interval $[t', t_2]$, p_l must have successfully observed the establishment of X prior to t_2. So it must then enter state $\langle success, X\rangle$. By Lemma 9 and Theorem 10, the processes in $P(X)$ will start an instance of X when their monitoring windows at t_2 expire. □

For the fairness property, again we need some assumption on the faultless behavior of the system. Unlike in the message-passing paradigm, no physical communication link for delivering messages is present between every pair of processes in the shared-memory model. So we need only to assume that processes are not hanging.

Theorem 13 (Weak interaction fairness). *Assume that processes are not hanging. If X is enabled at time t then, with probability* 1, *X will be disabled eventually.*

Proof. The proof is similar to Theorem 5, and note that Lemma 12 and a lemma similar to Lemma 3 is needed for the proof. □

Theorem 14 (Strong interaction fairness). *Assume* (A1) *that processes are not hanging, and* (A2) *that a process's transition to a state ready for interaction does not depend on the random draws performed by other processes. If an interaction X is enabled infinitely often then, with probability* 1, *the interaction will be executed infinitely often.*

Proof. Similar to Theorem 6. □

The time complexity of SM can be analyzed as in Section 3.2.4.

5. Concluding remarks

We have proposed two randomized algorithms, one for message passing and the other for shared memory, that, with probability 1, schedule multiparty interactions in a strongly interaction fair manner. Both algorithms improve upon a previous result by Joung and Smolka in the following aspects: first, processes' speeds and communication delays need not be bounded by any predetermined constant; second, the algorithms are completely decentralized, and the shared-memory solution makes use of only single-writer variables; and third, the algorithms are symmetric in the sense that all processes execute the same code, and no unique identifiers are used to distinguish processes.

In algorithm TB, a process p_i attempting to establish X adjusts its Δ based on the length of non-monitoring windows sent by the other processes in $P(X)$. Suppose for each $p_j \in P(X)$, the maximum length of p_j's non-monitoring window known by p_i is less than η_j. As we have shown, the necessary condition for TB to satisfy the fairness requirement is that $\Delta \geqslant \sum_{p_j \in P(X)-\{p_i\}} \eta_j$. Since Δ and each η_j are measured by different processes using their own clocks, in the algorithm we have assumed that processes' clocks tick at the same rate. Clearly, if the clocks may move at different

rate, then the condition $\Delta \geqslant \sum_{p_j \in P(X)-\{p_i\}} \eta_j$ (where the interpretation of Δ and η_j is with respect to a universal clock) may no longer be satisfied. However, if the relative clock speed between p_i and p_j is known, then p_i can time η_j by the drift rate to compensate its reading of η_j. If such a factor is not available, then, since a temporary choice of a short Δ cannot cause the algorithm to err, p_i can incrementally enlarge its Δ so that eventually the condition $\Delta \geqslant \sum_{p_j \in P(X)-\{p_i\}} \eta_j$ will be met. The situation is similar for algorithm SM.

Both algorithms cannot tolerate *zero-speed failure*, meaning that a process can stop prematurely (without forging or corrupting any of its variables). For algorithm TB, a process's failure may stop the whole system. This is because if a process p_j fails, then any process p_i which attempts to establish an interaction with p_j may have already sent its token to p_j and is waiting for p_j's acknowledgment or its return of the token. It is well known that, under the assumption of unbounded communication delay, p_i cannot distinguish whether p_j has already terminated, or has not yet responded to p_i's request. So, p_j's failure may hang p_i, which in turn will also hang all other processes waiting for p_i's response, and so on.

For algorithm SM, if p_j fails after it has expressed its interest in X (by setting $p_j.state$ to $\langle examining, X\rangle$), then p_i could establish X by changing $p_i.state$ to $\langle success, X\rangle$, and then waits forever in line 17 of SM for p_j to complement $p_j.flag[X]$. Note, however, that unlike TB, the other processes not involved in $P(X)$ may still be able to proceed in this situation. This is because another process p_k attempting to establish an interaction, say Y, waits for the other participants only in a bounded Δ-interval, and it learns their states by actively reading their variables. So, if Y also involves p_i (which has been trapped in an indefinite loop waiting for X to be established), then p_k will eventually time-out its Δ to give up on Y because not all processes in $P(Y)$ are interested in Y. So, p_k will be able to re-try another interaction. Of course, if no other interaction involving p_k is enabled, then p_k will also be blocked from establishing an interaction, even though some interaction involving p_k (e.g., Y) has been enabled.

It should be pointed out that, although in general the cost of randomized algorithms is considerably high, they may still out-perform existing deterministic algorithms (where only WIF is required) if response time is a main concern and the two parameters, k — the number of potential interactions for which a process may be ready at a time, and m — the number of participants in an interaction, can be kept small relative to n—the total number of processes in the system. Even if the above conditions cannot be met, randomized algorithms still have a niche because no deterministic algorithms are able to claim SIF.

Finally, we note that the fairness property of both algorithms is based on two assumptions. For weak interaction fairness, we require Assumption (A1) that a process cannot be hanging in the sense its speed cannot reduce to zero and there cannot exist an infinite sequence of steps of the process such that the lengths of the steps are monotonically increasing. For strong interaction fairness, we additionally require Assumption (A2) that a process's transition to a state ready for interaction does not depend on the random choices performed by other processes (so that two random draws

by different processes are always independent). It remains open whether either assumption can be removed. However, by observing the impossibility phenomena of strong interaction fairness in a deterministic setting [32, 16] and by the example discussed after Theorem 6, we conjecture that Assumption (A2) cannot be removed from strong interaction fairness.

Acknowledgements

The author would like to thank Jen-Yi Liao for his helpful discussion. The algorithms presented in the paper are based on some early discussion with Jen-Yi Liao. The author would also like to thank the anonymous referees for their valuable comments and suggestions, and David Aldous for pointing out the reference on the second Borel–Cantelli Lemma. This research was supported in part by the National Science Council, Taipei, Taiwan, under Grants NSC 84-2213-E-002-005, NSC 85-2213-E-002-059, and NSC 86-2213-E-002-053, and by the 1997 Research Award of College of Management, National Taiwan University.

References

[1] Y. Afek, H. Attiya, A. Fekete, M. Fischer, N.A. Lynch, Y. Mansour, D.-W. Wang, L. Zuck, Reliable communication over unreliable channels, JACM 41(6) (1994) 1267–1297.
[2] K.R. Apt, N. Francez, S. Katz, Appraising fairness in languages for distributed programming, Distributed Comput. 2(4) (1988) 226–241.
[3] R.J.R. Back, R. Kurki-Suonio, Distributed cooperation with action systems, ACM Trans. Programming Languages Systems 10(4) (1988) 513–554.
[4] R.L. Bagrodia, Process synchronization: design and performance evaluation of distributed algorithms, IEEE Trans. Software Eng. SE-15(9) (1989) 1053–1065.
[5] R.L. Bagrodia, Synchronization of asynchronous processes in CSP, ACM Trans. Programming Languages Systems 11(4) (1989) 585–597.
[6] P. Billingsley, Probability and Measure, 3rd ed., Wiley Series in Probability and Mathematical Statistics, Wiley, New York, 1995.
[7] T. Bolognesi, E. Brinksma, Introduction to the ISO spec. language LOTOS, Comput. Networks ISDN Systems 14 (1987) 25–59.
[8] G.N. Buckley, A. Silberschatz, An effective implementation for the generalized input–output construct of CSP, ACM Trans. Programming Languages Systems 5(2) (1983) 223–235.
[9] K.L. Chung, A Course in Probability Theory, 2nd ed., A Series of Monographs and Textbooks, Academic Press, New York, 1974.
[10] N. Francez, I.R. Forman, Interacting Processes: A Multiparty Approach to Coordinated Distributed Programming, Addison-Wesley, Reading, MA, 1996.
[11] N. Francez, B. Hailpern, G. Taubenfeld, Script: a communication abstraction mechanism, Sci. Comput. Programming 6(1) (1986) 35–88.
[12] N. Francez, M. Rodeh, A distributed abstract data type implemented by a probabilistic communication scheme, Technical Report TR-80, IBM Israel Scientific Center, April 1980, A preliminary version appeared in the Proc. 21st Ann. IEEE Symp. on Foundations of Computer Science, Long Beach, California, 1980, pp. 373–379.
[13] C.A.R. Hoare, Communicating sequential processes, Commun. ACM 21(8) (1978) 666–677.
[14] H.-M. Järvinen, R. Kurki-Suonio, DisCo specification language: marriage of actions and objects. Proc. 11th Int. Conf. on Distributed Computing Systems, Arlington, TX, May 1991, IEEE Computer Society Press, Silver Spring, MD, pp. 142–151.

[15] H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen, K. Systä, Object-oriented specification of reactive systems, Proc. 12th Int. Conf. on Software Engineering, Nice, France, March 1990, IEEE Computer Society Press, Silver Spring, MD, pp. 63–71.
[16] Y.-J. Joung, Characterizing fairness implementability for multiparty interaction, Proc. 23rd Int. Colloquium on Automata, Languages and Programming, Paderborn, Germany, July 1996, Published by Lecture Notes in Computer Science, vol. 1099, Springer, Berlin, pp. 110–121.
[17] Y.-J. Joung, S.A. Smolka, Coordinating first-order multiparty interactions, ACM Trans. Programming Languages Systems 16(3) (1994) 954–985.
[18] Y.-J. Joung, S.A. Smolka, A comprehensive study of the complexity of multiparty interaction, J. ACM 43(1) (1996) 75–115.
[19] Y.-J. Joung, S.A. Smolka, Strong interaction fairness via randomization, Proc. 16th Int. Conf on Distributed Computing Systems, Hong Kong, May 1996, pp. 475–483.
[20] D. Kumar, An implementation of N-party synchronization using tokens, Proc. 10th Int. Conf on Distributed Computing Systems, Paris, France, 28 May–1 June 1990, pp. 320–327.
[21] D. Lehman, M.O. Rabin, On the advantage of free choice: a symmetric and fully distributed solution to the dining philosophers problem (extended abstract). Proc. 8th Ann. ACM Symp. on Principles of Programming Languages, ACM Press, New York, 1981, pp. 133–138.
[22] N.A. Lynch, Distributed Algorithms, Morgan-Kaufmann, Los Altos, CA, 1996.
[23] R. Milner, Calculi for synchrony and asynchrony, Theoret. Comput. Sci. 25 (1983) 267–310.
[24] R. Milner, Communication and Concurrency, International Series in Computer Science, Prentice-Hall, United Kingdom, 1989.
[25] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, I. Inform. Comput. 100(1) (1992) 1–40.
[26] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, II. Inform. Comput. 100(1) (1992) 41–77.
[27] M.H. Park, M. Kim, A distributed synchronization scheme for fair multi-process handshakes, Inform. Process. Lett. 34 (1990) 131–138.
[28] S. Ramesh, A new and efficient implementation of multiprocess synchronization, Proc. Conf. on PARLE, Lecture Notes in Computer Science, vol. 259, Springer, Berlin, 1987, pp. 387–401.
[29] S. Ramesh, A new efficient implementation of CSP with output guards, Proc. 7th Int. Conf. on Distributed Computing Systems, Berlin, Germany, 1987, pp. 266–273.
[30] J.H. Reif, P.G. Spirakis, Real time synchronization of interprocess communications, ACM Trans. Programming Languages Systems 6(2) (1984) 215–238.
[31] P.A. Sistla, Distributed algorithms for ensuring fair interprocess communications, Proc. 3rd Ann. ACM Symp. on Principles of Distributed Computing, ACM Press, New York, 1984, pp. 266–277.
[32] Y.-K. Tsay, R.L. Bagrodia, Some impossibility results in interprocess synchronization, Distributed Comput. 6(4) (1993) 221–231.
[33] Y.-K. Tsay, R.L. Bagrodia, Fault-tolerant algorithms for fair interprocess synchronization, IEEE Trans. Parallel Distributed Systems 5(7) (1994) 737–748.
[34] U.S. Department of Defense, Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, U. S. Government Printing Office, Washington, DC, January 1983.