Fairness of N-party Synchronization and Its Implementation in a Distributed Environment[1]

Cheng Wu, Gregor v. Bochmann, Mingyu Yao

Departement d'informatique et de recherche operationnelle
Universite de Montreal, Montreal, P.Q., Canada

Abstract Fairness is an important concept in design and implementation of distributed systems. At the specification level, fairness usually serves as an assumption for proving liveness. At implementation level, the question becomes how to implement the underlying fairness which is assumed to be true at the specification level. In this paper, we study four types of fairness, the so-called w-fairness (weak fairness), s-fairness (strong fairness), u-fairness and su-fairness, in the context of the design of N-party synchronization algorithms. Within an abstract model for distributed systems, we formally introduce the four fairness concepts. We formally present, in the form of extended finite state machines, several distributed N-party synchronization algorithms which satisfy different fairness properties. The algorithms given in this paper are abstract in a sense that they are not optimized. The abstraction makes the construction of the algorithms and their proof of correctness easier.

1. Introduction

Fairness is an important property in the design and implementation of distributed systems. When specifying a distributed system, one usually assumes that the system has some kind of fairness. And based on these fairness assumptions, liveness properties can be proved [4]. Liveness properties are usually described as "good things will eventually happen" and fairness properties are usually described as "if something is always or infinitely often ready then it will eventually happen". So, if one can show that a "good thing" meets some conditions such as being always or infinitely often ready, then fairness assumptions will lead to the related liveness property, that is, "it will eventually happen". It is clear that formal definitions of fairness are needed to prove liveness. There is a lot of work in this area (see for instances [4, 7, 13]).

Besides obtaining a proof system, another motivation for studying fairness properties is to implement fairness, that is, to make sure that the implementation of a distributed system has the fairness properties which are assumed to be true in its specification. Thus properties which are proved to hold at the specification level based on those fairness assumptions, will also hold for the implementation. In this paper, we mainly address the question of implementing N-party synchronization with different kinds of fairness proprieties. We are interested here in four fairness properties, namely w-fairness, s-fairness, u-fairness and su-fairness. Here, w-fairness and s-fairness are the well known weak fairness and strong action (interaction)

[1] This work was performed within the IDACOM-NSERC-CWARC Industrial Research Chair on Communication Protocols and was also supported by the Ministry of Education of Quebec.

fairness, respectively; u-fairness is defined in [1] and is also called handshake fairness in [2]; su-fairness is introduced in this paper to implement s-fairness.

N-party synchronization (also called multi-way rendezvous) is an important concept [3], and is used in certain specification languages such as LOTOS [6]. In N-party synchronization, an arbitrary number of processes may synchronize. The question of implementation of N-party synchronization respecting the fairness properties has been studied by various authors. The problem is hard for a distributed solution, where each process only has a partial picture of the system's global state. Distributed solutions for w-fairness were presented in [11, 8] and one for u-fairness was presented in [1]. S-fairness is proven by the work of [12] to be impossible to implement based on an assumption that an active process may not become idle to communicate with other processes (that is, a process may always do local (internal) actions). In fact, under this assumption, the u-fairness is also impossible to implement, which will be discussed later in this paper.

Our work presented in this paper is based on an assumption that any action will eventually terminate after it starts (that is, an active process will eventually become idle to communicate with other processes). In this paper, we first define an abstract model for distributed systems. Then we give definitions of the four fairness properties above by using the formalisms of transition systems and linear temporal logic. Su-fairness introduced in this paper is stronger than s-fairness. We aim to implement su-fairness instead of s-fairness. An algorithm which satisfies su-fairness also satisfies s-fairness.

We present four distributed algorithms for N-party synchronization. Algorithms 1 and 2 satisfy w-fairness and u-fairness respectively. Algorithm 3 satisfies u-fairness, and Algorithm 4 satisfies s-fairness and su-fairness. We do not give any algorithm which satisfies exactly s-fairness. The algorithms presented in this paper are abstract in a sense that they are not optimized. The abstraction makes the construction of the algorithms and their proof of correctness easier. In fact, in this paper, Algorithm **i** is constructed based on Algorithm **j** ($j < i$) and the correctness proof of the former is based on the properties of the latter.

This paper is organized as follows. In Section 2, we give the abstract model for distributed systems and define w-fairness, s-fairness, u-fairness, and su-fairness. In Section 3, we present the four distributed algorithms for N-party synchronization which satisfy different fairness properties. Finally, Section 4 contains our conclusions.

2. An abstract model for distributed systems and fairness definitions

2.1. Theoretical framework - linear Temporal Logic and labeled transition systems

Fairness properties are related to infinite execution histories. Usually so-called "weak fairness" and "strong fairness" properties are described informally as "if permanently (always) A then eventually B" and, respectively, "if infinitely often A then eventually B". Temporal Logic can be used to describe temporal concepts such as *permanently(always)*, *infinitely often* and *eventually* [4, 7, 9]. In the following we will give a brief introduction to Linear Temporal Logic [10] which will be used to define fairness properties in this paper.

Besides the ordinary logical operators **or**, **and**, **not**, and □ (imply), Linear Temporal Logic uses two temporal operators □ and []. The expression [] P (read "henceforth P") means that P is true now and will always be true in the future, and □ P (read "eventually P") means that P is true now or will be true sometimes in the future. They are usually interpreted based on a computation model of state sequences. The temporal concepts mentioned above can be described by the two temporal operators and their combinations. For example, □[] P means from a certain time onwards permanently P; []□ P means infinitely often P.

In Section 2.2 we will use Labeled Transition Systems to describe our model for distributed systems. In the following, we show several interesting concepts related to Labeled Transition Systems, which will be used to define fairness properties.

Definition 2.1 A *labeled transition system* is a triple **LTS**=(**S**,**T**,{-**t**->} $_{\mathbf{t□T}}$) where,
- **S** is a countable set of states
- **T** = {**t1**, **t2**, ...} is a set of labeled transitions
- {-**t**->}$_{\mathbf{t□T}}$ is a set of binary relations on S (S∞S□ -**t**->) in bijection with the labeled transitions.

Definition 2.2. An *execution history* **h** is a sequence of transition states $\mathbf{s_0(\varepsilon),s_1(t_1),s_2(t_2),...}$, where $\mathbf{s_i}$□**S**, $\mathbf{t_i}$□**T** and <$\mathbf{s_{i-1}}$,$\mathbf{s_i}$>□ -$\mathbf{t_i}$->, and $\mathbf{s_0(\varepsilon)}$ is an initial transition state. Note: each transition state $\mathbf{s_i(t_i)}$ of an execution history represents the state $\mathbf{s_i}$ of the LTS and the transition $\mathbf{t_i}$ that was "executed" by the system when entering this state.

Definition 2.3. For a given transition state **s(t)** in an execution history **h**, a transition labelled **t'** is said to be *executed*, which is denoted as EXECUTED(**t'**), iff **t = t'**.

Note, EXECUTED is an assertion on transition states. We can combine the assertion EXECUTED with the two temporal operators [] and □. For instance, []□ EXECUTED(**t**) means transition **t** is **infinitely often** executed in an execution history.

2.2. A model for distributed systems

We consider here a distributed system consisting of a fixed set of processes and a fixed set of actions. Each action is associated with a non-empty set of processes which is called its *process-set*. Each process participates in a non-empty set of actions which is called its *action-set*. The process-sets and the action-sets are static, that is, their membership does not change.

Processes are executed concurrently. A process executes one action at a time. Processes have disjoint local state spaces, that is, there are no shared variables. A process contains a set of local variables. One of them is called *control state variable* which records the control state of the process. At any instance in time, the control state vairable of a process has one of the following two values: *idle* which denotes that the process is waiting to execute actions (we say that the process is in the *idle state*); *execution* which denotes that the process is executing an action (we say that the process is in the *execution state*). In executing an action, a process changes some of its local variables. A process associates each of the actions in its action-set with a *guard*, a boolean function on its local variables (excluding the control state variable); it is said to be willing to do an action if the guard for the action is true. A process can autonomously change from the execution state to the idle state when the execution of an action terminates. We assume that a process takes finite time to finish the execution of any action, which ensures that a process eventually changes form the execution state to the idle state whenever it is in the execution state. In order to enter the execution state, a process has, in general, to communicate with the other processes that also participate in the execution of the action. A process immediately changes from the idle state to the execution state when it commits to execute an action.

An action can only be ready to execute when all the processes in its process-set are in their idle state and willing to do the action, that is, related guards of the action are true. An action is called a "communication", a "channel" or an "inter-action" if its process set has more than one element. An action is called a"local action" if its process set includes only one process.

To clearly define fairness, we need to formally describe the model above. We use labelled transition systems. We first consider that the behavior of each process in the system is given by a labelled transition system. Then the behavior of the system is defined by a global labelled transition system which is built from the local transition systems of the processes in the system.

We consider here that the execution of an action takes certain time and, after committing to execute an action, a process will not be ready for the execution of another action for some time. To model this, we assume that a process spends some time in the execution state after entering it. We also assume that the execution of an action will eventually terminate, thus a process will eventually return to its idle state. In the following, we introduce a silent transition **i** to explicitly denote the termination of executing an action (we make no difference between the termination of different actions). In the following, the execution of a transition **a** denotes the beginning of the execution of the action **a** (by all related processes), while the execution of the silent transition **ip** denotes the termination of the participation of process **p** in the execution of the last action. The interleaving model of the state transition system therefore allows the modelling of actions being executed concurrently.

Let **P={p1,...,pn}** be the set of processes in the system and **A={a1,...,am}** be the set of actions in the system. Let **Pa** (**P□Pa**) denote the process-set of action **a** (**a□A**) and **Ap** (**A□Ap**) denote the action-set of process **p** (**p□P**). Let **Sp** denote the state set of process **p** where a state of a process defines values of its local variables. Let **ip** denote a silent action of the process **p**, which denotes the termination of the execution of any action (since we make here no difference between the termination of executing different actions). Let **state-type**$_{\mathbf{p}}$ and **guard**$_{\mathbf{p}}$ be two functions on states of the process **p**, where **state-type**$_{\mathbf{p}}$**:Sp -> {idle, execution}** denotes the type of a state of process **p** (i.e., the value of the control state variable of process **p**) and **guard**$_{\mathbf{p}}$**:Sp∞Ap-> {true, false}** denotes whether process **p** in a given state is willing to participate in an action.

Then we assume that the behavior of process **p** (**$p \in P$**) is given by a labelled transition system **$LTS = (Sp, Ap \cup \{ip\}, \{\text{-}t\text{->}\}_{t \in Ap \cup \{ip\}})$**. A transition **-t->** ($t \in Ap$) denotes the execution of the action **t**. It changes the control state variable from 'idle' to 'execution'. It may change some of the local variables and thus it may also change some local guards. The transition **-ip->** denotes the local termination of an action execution. It changes the control state variable from 'execution' to 'idle' and it does not change any other local variable. For a process in the model above, each idle state follows an execution state and each execution state follows an idle state (see the above sub-section). Therefore, the transition system of a process has properties which can be described formally as the follows:

(1) **s' -t-> s"** iff **state-type_p(s')=idle** and **state-type_p(s")=execution** and **guard_p(s',t)** and **$s', s" \in Sp$** and **$t \in Ap$**;
(2) **s'-ip->s"** iff **state-type_p(s')=execution** and **state-type_p(s")=idle** and **$s', s" \in Sp$**
(3) **guard_p(s', a) = guard_p(s", a)** for any **$a \in Ap$**
if **s'-ip->s"** (the assumption of that the execution of silent action ip does not change any local variable)

Let **$S = S_{p1} \times S_{p2} \times \ldots \times S_{pn}$** be the set of global states. Let **$I = \{ip | p \in P\}$**. Then the behavior of the distributed system is defined by a label transition system **$LTS=(S, A \cup I, \{\text{-}t\text{->}\}_{t \in A \cup I})$**, which has the following properties:

(1) $<s_{p1}', s_{p2}', \ldots, s_{pn}'>$ **-t->** $<s_{p1}", s_{p2}", \ldots, s_{pn}">$
iff **$t \in A$** and s_{pi}'-t-> $s_{pi}"$ for each **$pi \in Pt$** and $s_{pj}'=s_{pj}"$ for each **$pj \in Pt$**;
(2) $<s_{p1}', s_{p2}', \ldots, s_{pi}', \ldots, s_{pn}'>$ **-i_{pi}->** $<s_{p1}', s_{p2}', \ldots, s_{pi}", \ldots, s_{pn}'>$
iff **$i_{pi} \in I$** and s_{pi}'-i_{pi}->$s_{pi}"$

The followings are predicates on global states which will be used later to define fairness. **IDLE_a** indicates whether all processes in the process-set of action **a** are in their idle state. **GUARD_a** indicates whether all processes in the process-set for action **a** have their (local) guards being **true** for **a**. **ENABLED_a** indicates whether the system is ready to execute action **a**, which means that all processes in the process-set of **a** have their guard being true (concerning **a**) and are in their idle states.

Definition 2.4. For a given **$a \in A$**, **IDLE_a** is a predicate on a state. For a given state **$s=<s_{p1}, s_{p2}, \ldots, s_{pn}>$**, **$\text{IDLE}_a$** = **true** iff **$\text{state-type}_{pi}(s_{pi})$=idle** for all **$pi \in Pa$**.

Definition 2.5. For a given **$a \in A$**, **GUARD_a** is a predicate on a state. For a given state **$s=<s_{p1}, s_{p2}, \ldots, s_{pn}>$**, **$\text{GUARD}_a$** = **true** iff **$\text{guard}_{pi}(s_{pi}, a)$** for all **$pi \in Pa$.**

Definition 2.6. For a given **$a \in A$**, **ENABLED_a** is a predicate on a state. **ENABLED_a= true** iff **IDLE_a** and **GUARD_a**.

2.3. Fairness properties

In this section we define four fairness properties, namely, w-fairness, s-fairness, u-fairness and su-fairness. The following definitions are in the context of the model of Section 2.2.

Definition 2.7 (w-fairness) An infinite execution history **h** respects w-fairness if it satisfies $\forall\, a \in A,\ [\,]\,(\Diamond[\,]\ \mathrm{ENABLED}_a \Rightarrow \Diamond\ \mathrm{EXECUTED}(a))$

W-fairness (also called weak fairness) describes that an action will eventually be executed if all related processes remain in their idle state and their guard concerning the action are true.

Definition 2.8 (s-fairness) An infinite execution history **h** respects s-fairness if it satisfies $\forall\, a \in A,\ [\,]\,([\,]\Diamond\ \mathrm{ENABLED}_a \Rightarrow \Diamond\ \mathrm{EXECUTED}(a))$

S-fairness (also called strong fairness) describes that an action will eventually be executed if all related processes are infinitely often ready together to execute the action, that is, they infinitely often synchronize at their idle state and their guard concerning the action are true. W-fairness and s-fairness are generally accepted fairness concepts [4].

Definition 2.9 (u-fairness) An infinite execution history **h** respects u-fairness if it satisfies $\forall\, a \in A,\ [\,]\,(\Diamond[\,]\ \mathrm{GUARD}_a \Rightarrow \Diamond\ \mathrm{EXECUTED}(a))$

U-fairness describes that an action will eventually be executed if the guards concerning the action always hold for all processes in its process-set. U-fairness was defined in [1] and is said to be good for the detection of stable properties in design of distributed system (see [1]).

Definition 2.10 (su-fairness) An infinite execution history **h** respects su-fairness if it satisfies $\forall\, a \in A,\ [\,]\,([\,]\Diamond\ \mathrm{GUARD}_a \Rightarrow \Diamond\ \mathrm{EXECUTED}(a))$

Su-fairness describes that an action will eventually be executed if the guards for the action infinitely often hold simultaneously for all processes in its process-set. We have introduced su-fairness here for the implementation of s-fairness, as discussed later. For the overlapping (concurrent) model of Section 2.2, s-fairness allows a so-called "conspiracy" phenomenon: An action may never be executed even if the guards for the action are always true for each process in its process-set. This may occur if the processes in the process-set of the action never synchronize on their idle states. Su-fairness and U-fairness do not allow such "conspiracy"; they implicitly require that processes synchronize on their idle-states, since this is necessary for the execution of an action.

The following theorem shows the relations among these fairness properties. S-fairness and u-fairness are not comparable. An execution history which respects s-fairness does not necessarily respect u-fairness, or vice-versa (see examples in [1]).

Theorem 2.1
(1) su-fairness □ s-fairness □ w-fairness
(2) su-fairness □ u-fairness □ w-fairness

Proof: In logic, we have (A □ B) □ ((B □ C) □ (A □ C)) for any formulas A, B and C. We have □[] ENABLEDa □ []□ ENABLEDa □ []□ GUARDa and □[] ENABLEDa □ □[] GUARDa □ []□ GUARDa The conclusions are immediate from the related fairness definitions.

3. Implementation of N-party synchronization respecting various fairness properties

In the section, we will describe four distributed algorithms. Algorithms 1 and 2 respect w-fairness and u-fairness, respectively. Algorithm 3 is a solution for both w-fairness and u-fairness. Algorithm 4 is designed to satisfy su-fairness and s-fairness. In the model described in Section 2.2, we assume that any action will eventually terminate after it starts. Without this assumption, it is impossible to implement s-fairness, as proved in [12], as well as u-fairness. For example, let **p** be a process which will not become idle. Then any interaction related to **p** can not be executed even if its guard is always true for all processes (including **p**) in its process-set.

In the following, we consider that the implementation of N-party synchronization is provided by a system consisting of two kinds of processes: *P-processes* and *A-processes*. For each process in the model of Section 2.2, there is a corresponding P-process. When a process in the model of Section 2.2 reaches its idle state, it informs its corresponding P-process and provides a list of (locally) possible actions (i.e., actions whose guards are true). When a P-process has found out that an interaction (global synchronization) is possible, it informs its corresponding process to enter its execution state to execute the interaction. Note that now, N-party synchronization means that a group of P-processes synchronize for the execution of an action.

For each action in the model of Section 2.2, there is a corresponding A-process in the system. The function of an A-process is to control the synchronization of P-processes. Processes (P-processes and/or A-processes) in the system communicate with each other asynchronously and each process has a FIFO queue to store the input messages. Each process has a unique identifier and processes are ordered by their identifiers. We assume "minimum liveness" for processes in the following algorithms. That is, a process does nothing only if it is in a dead-lock state. The proof of correctness of the following algorithms in based on this assumption.

3.1. Algorithm 1

3.1.1. Informal description

An A-process can be in two states: inactive and active. An A-process changes its state spontaneously and in a finite time from inactive to active. When in the active state, an A-process of an action **a** tries to capture all P-processes in the process-set of the action. To capture a P-process, an A-process sends a *capture* message to the P-process. Afterwards, it either receives a *yes* message from the latter, which means that the capture of the P-process succeeded, or receives a *no* message, which means that the capture of the P-process failed. An A-process captures P-processes one by one in the increasing order of their identifiers. That is, an A-process can only try to capture a P-process **p** if it has captured all P-processes whose identifiers are smaller than **p** in the process-set of the action. If all processes in the process-set of action **a** are captured, the A-process of action **a** sends each related P-process a *commit* message and the latter commits to execute the action. After that, the A-process changes its state to inactive. If a capture procedure failed due the reception of a *no* message from a P-process, the A-process aborts the capture procedure by sending to each P-process which was captured previously an *undo* message. Then it changes its state to inactive.

Capture messages to a P-process are first queued in its *capture-message-queue* (which is FIFO queue). The P-process de-queues and processes these messages one by one according to its internal state.

A P-process can be in one of two states: idle or execution (which correspond with the idle and execution states, respectively, in the model of Section 2.2). After executing an action, a P-process spontaneously changes from its execution state to idle. When it commits to execute an action, a P-process changes from the idle state to the execution state. A P-process sends back a *no* message to an A-process if it de-queues (or processes) a *capture* message in its execution state. In its idle state, a P-process does the followings depending on the situation: It sends back a *no* message to the A-process if the local guard for the related action is false. If it is not captured by another A-process and the local guard for the related action is true, the P-process agrees to be captured and sends back a *yes* message. After being captured, it either executes an action upon receiving the *commit* message, or becomes not captured upon receiving the *undo* message. The P-process delays the processing of any additional *capture* messages when it is captured.

In the following formal description, the behavior of a P-process is specified by two machines M1 and M2. M1 and M2 communicate through a queue (i.e., capture_message_queue). The idle state of a P-process is modeled by two states: Idle_free (to model that the P-process is not captured) and Idle_captured (to model that the P-process is captured).

3.1.2. Formal description

In the following we will use extended finite state machines to formally describe the algorithm. We assume here A-processes and P-processes communicate through reliable message communications. We also assume that an A-process has as identifier the name of its related action.

(A) The types of messages transmitted between A- and P-processes

capture(a, p): a message from A-process **a** to P-process **p** (about action **a**), which means that **a** tries to capture **p** for a rendezvous concerning action **a**;

yes(p, a): a message from P-process **p** to A-process **a**, which means that **p** agrees to be captured by **a**;

no(p, a): a message from P-process **p** to A-process **a**, which means that **p** does not agree to be captured by **a**;

commit(a, p): a message from A-process **a** to P-process **p**, by which **a** asks **p** to execute action **a**;

undo(a, p): a message from A-process **a** to P-process **p**, by which **a** indicates to **p** that the current capture procedure concerning action **a** is abandoned.

(B) The behavior of an A-process

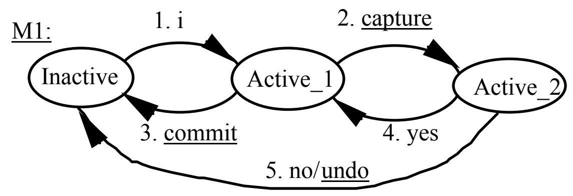
Constants

id :	the identifier of the process;
p_list :	a list of process identifiers (obtained by sorting the process-set of the action), p_list(i) denotes the i-th element of the p_list;
p_len :	the length of p-list.

Variables

p_index: 1.. p_len + 1 :	to record the process which is to be or being captured.
i: 1.. p_len .	

State graph



Transition table

Transitions	*Conditions*	*Actions*
1. i		p_index := 1
2. capture	**if** p_index ≤ p_len	**send** capture(id, p_list(p_index))
3. commit	**if** p_index > p_len	**for** i := 1 **to** p_len **do** **send** commit(id, p_list(i))
4. yes	**if** yes(p,id) **is received**	p_index := p_index + 1
5.no/undo	**if** no(p,id) **is received**	**for** i := 1 **to** p_index-1 **do** **send** undo(id, p_list(i))

Initialization

M1 is in the Inactive state.

(C) The behavior of a P-process

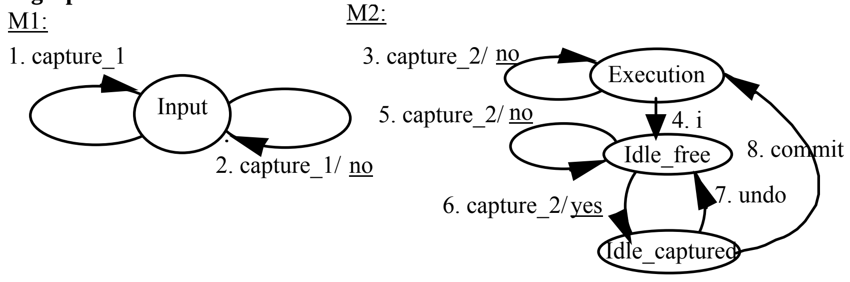
Constants

id :	the identifier of the process.

Variables

guard(a) :	the guard of action **a**;
capture_message_queue :	the queue in which received *capture* message are stored

State graph



Transition table

Transitions	*Conditions*	*Actions*

1. capture_1	**if** capture(a, id) **is received and** guard(a)	**put** capture(a, id) **into** capture_message_queue
2. capture_1/<u>no</u>	**if** capture(a,id) **is received and not** guard(a)	**send** no(id, a)
3. capture_2/<u>no</u>	**if get** capture(a, id) **from** capture_message_queue	**send** no(id, a)
4. i		
5. capture_2/<u>no</u>	**if get** capture(a, id) **from** capture_message_queue **and not** guard(a)	**send** no(id, a)
6. capture_2/<u>yes</u>	**if get** capture(a, id) **from** capture_message_queue **and** guard(a)	**send** yes(id, a)
7. undo	**if** undo(a, id) **is received**	
8. commit	**if** commit(a, id) **is received**	(to execute action **a)**

Initialization
M1 is in the Input state and M2 is in the Execution state; Capture_message_queue is empty.

3.1.3. Discussion

The proof of correctness of Algorithm 1 is given in [14]. It is easy to see that Algorithm 1 satisfies synchronization and mutual exclusion required by N-party synchronization. In the algorithm, an A-process sends out a *commit* message only if it captures all the processes in the process-set of the action. A P-process agrees to be captured only if it is in its idle state and the guard of the action holds and it can only be captured once at a time.

Algorithm 1 satisfies w-fairness. In the algorithm, a capture message concerning an action which is always enabled will eventually be processed by related P-processes. This is so because each process satisfies minimum liveness and the message is stored in FIFO queues.

3.2. Algorithm 2

Algorithm 2 is designed to satisfy u-fairness. It is obtained by modifying Algorithm 1. The idea is to force P-processes to synchronize on their idle-states. Instead of responding *no* when it receives any *capture* message in its execution state, a P-process postpones the processing of *capture* messages which are received in its execution state until it reaches its idle state. The formal description is the same as the one in Section 3.1.2., except that transition 3 of the P-process is deleted.

The proof of correctness of Algorithm 2 is given in [14]. Algorithm 2 satisfies u-fairness. This is because a *capture* message will never be discarded by P-processes unless its guard is false and P-processes are forced to synchronized on their idle-state.

3.3. Implementation of s-fairness and su-fairness

We present here two N-party synchronization algorithms. The first one, Algorithm 3, respects u-fairness. It is constructed based on the Algorithms 2 above. The second one, Algorithm 4, is designed to implement su-fairness. From Theorem 2.1, we know that an algorithm satisfies s-fairness if it satisfies su-fairness. Algorithm 4 is obtained by modifying Algorithm 3.

For the model of Section 2.2, two actions **a** and **b** are said to be conflicting if they share some process, that is, Pa ↔ Pb ≠ ϕ (where Pa and Pb are the process-sets of **a** and **b**, respectively). To facilitate the description in the following, we call Ca = {b | Pa↔Pb ≠ ϕ} the conflict-set of the action **a** (note, a□Ca).

3.3.1. Algorithm 3

The general idea of this algorithm is as follows: In Algorithms 1 and 2, a P-process can only be captured by one A-process at a time, which ensures mutual exclusion. In Algorithm 3, mutual exclusion is ensured by the A-processes. That is, an A-process **a** starts a capture procedure only after making sure that all conflicting A-processes **b** (b□Ca) do not start their respective capture procedures. Therefore a P-process, after receiving a *capture* message, will receive either a *commit* message or an *undo* message, but not another *capture* message. If Algorithm 3 can ensure that all A-processes in the system infinitely often start their capture procedure, then it is easy to show that the algorithm respects u-fairness if the P-processes behave as defined in Algorithm 2. The reason is that a capture procedure succeeds whenever it starts if the related guards are always true.

Algorithm 3 can be easily constructed based on Algorithm 2. As described above, A-processes in Algorithm 3 work in two phases, a phase of synchronization among A-processes and a phase of capturing P-processes. We consider that each Ca (for all a□A) defines a rendezvous relation among A-processes and the execution of a rendezvous Ca means that A-process **a** has the right to start its capture procedure and the other A-processes in Ca do not start their capture procedures. We use Algorithm 2 to synchronize the A-processes. We design the A-processes such that they are always willing to participate in any rendezvous Ca (this means that all local guards for the rendezvous Ca are always true). Then from Theorem 3.2, each rendezvous will be executed infinitely often, that is, each A-process will infinitely often start its capture procedure. The capture procedure will be the same as in Algorithm 2.

3.3.2. Algorithm 4

The key idea of Algorithm 4 is to introduce priority among actions. The action which has the highest priority will be executed whenever its guard is true. Algorithm 4 also forces processes to synchronize on their idle-states.

3.3.2.1. Informal description

Like Algorithm 3, Algorithm 4 also works in two phases: synchronizing A-processes and capturing P-processes. To implement su-fairness, we consider that different actions have different priorities. The priority of an action may change during the execution and the more an action is executed, the lower becomes its priority. During the phase of A-process synchronization, an A-process **a** tries to capture all A-processes (instead of capturing A-processes in Ca as in Algorithm 3) to ensure the priority. Let **Ra** denotes a rendezvous relation among all A-processes and the execution of a rendezvous **Ra** means that A-process **a** has the right to start its capture procedure and the other A-processes do not start their capture procedures. An A-process **a** is willing to participate in **Rb** only if the priority of **b** is higher than the one of **a**, or it is sure that the execution of action **a** is not possible at the moment (the previous capturing procedure concerning action **a** failed and no P-process **p** (p□Pa) has been in its execution state (which may have changed its state) ever since). Whenever an action **a** is executed, all A-processes are informed. The execution of the action **a** may change the state of P-processes **p** (p□Pa). Thus after knowing the execution of action **a**, an A-process **b** (b□Ca) which used to believe that the execution of action **b** is not possible does not have such confidence any more. Like for Algorithm 3 as described above, after a rendezvous **Ra** is executed, the A-process **a** starts its capture procedure, which is the same as in Algorithm 2.

3.3.2.2. Formal descriptions

(A) The types of messages

Between A-processes

capture_a(a, a'): : a message from A-process **a** to A-process **a'**, which means that **a** tries to capture **a'** for a rendezvous concerning the rendezvous relation **Ra**;

yes_a(a', a): a message from A-process **a'** to A-process **a**, which means that **a'** agrees to be captured by **a**;

no_a(a', a): : a message from A-process **a'** to A-process **a**, which means that **a'** does not agree to be captured by **a**;

commit_a(a, a'): : a message from A-process **a** to A-process **a'**, by which **a** informs **a'** about the execution of **Ra**;

undo_a(a, a'):: a message from A-process **a** to A-process **a'**, by which **a** indicates to **a'** the abortion of the current capture procedure concerning the rendezvous relation **Ra**.

Between A- and P-processes

capture_p(a, p): a message from A-process **a** to P-process **p**, which means that **a** tries to capture **p** for a rendezvous concerning action **a**;

yes_p(p, a): a message from P-process **p** to A-process **a**, which means that **p** agrees to be captured by **a**;

no_p(p, a): a message from P-process **p** to A-process **a**, which means that **p** does not agree to be captured by **a**;

commit_p(a, p): a message from A-process **a** to P-process **p**, by which **a** asks **p** to execute action **a**;

undo_p(a, p): a message from A-process **a** to P-process **p**, by which **a** indicates to **p** the abortion of the current capture procedure concerning action **a**.

(B) The behavior of an A-process

Constants

id :	the identifier of the process;
a_list :	a list of A-process identifiers (obtained by sorting all actions in the system), a_list(i) denotes the i-th element of the a_list;
conflict:	the conflict set of the action;
a_len :	the length of a_list;
p_list :	a list of P-process identifiers (obtained by sorting Pa, the process-set of the action), p_list(i) denotes the i-th element of the p_list;
p_len :	the length of p-list;

Variables

a_index: 1.. a_len+1
i_a : 1 .. a_len
captured_all_a : boolean : to record if the capturing of A-processes has succeeded
captured_all_p: boolean : to record if the capturing of P-processes has succeeded
tried: boolean :

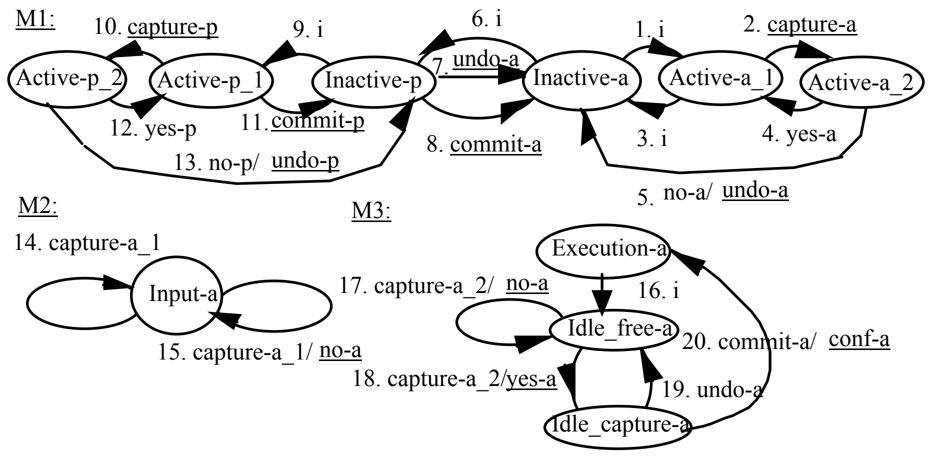
tried = true means that the latest capture P-process procedure failed;
tried = false means that the A-process needs to start a new procedure to capture P-processes.

action_priority_queue :

a list of actions. It supports the following two operations:
a > b: boolean : which is true if **b** is behind **a** in the queue;
decrease_priority(a) : which removes **a** from the current position and puts it at the tail of the queue

capture-a_message_queue : the queue in which received *capture-a* messages are stored
p_index: 1.. p_len+1 : to record the process which is to be or being captured.
ip : 1.. p_len

State graph



Transition table

Transitions	*Conditions*	*Actions*
1. i	**if not** tried **and not** captured_all_a	a_index := 1
2. capture-a	**if** a_index ≤ a_len	**send** capture-a(id,a_list(a_index))
3. i	**if** a_index > a_len	captured_all_a := true
4. yes-a	**if** yes-a(a',id) **is received**	a_index := a_index + 1
5. no-a/undo-a	**if** no-a(a',id) **is received**	**for** i_a := 1 **to** a_index-1 **do send** undo-a(id, a_list(i_a)); captured_all_a := false
6. i	**if not** captured_all_p **and** captured_all_a **and not** tried	
7. undo-a	**if** tried **and not** captured_all_p	**for** i_a := 1 **to** a_len **do** **send** undo-a(id, a_list(i_a)); captured_all_a := false

8. commit-a	**if** captured_all_p	**for** i_a:= 1 **to** a_len **do**
		send commit-a(id, a_list(i_a));
		captured_all_a := false;
		captured_all_p := false

9. i	**if not** tried **and not** captured_all_p	p_index := 1
10. capture-p	**if** p_index ≤ p_len	**send** capture-p(id, p_list(p_index))

11. commit-p	**if** p_index > p_len	captured_all_p := true
		for i_p := 1 **to** p_len **do**
		send commit-p(id,a_list(i_p));

12. yes-p	**if** yes-p(p,id) **is received**	p_index := p_index + 1
13. no-p/undo-p	**if** no-p(p,id) **is received**	**for** i_p := 1 **to** p_index-1 **do send** undo-p(id, p_list(i_p)); tried := true
14. capture-a_1	**if** capture-a(a',id) **is received and** (tried **or** a' > id)	**put** capture-a(a', id) **into** capture-a_message_queue
15. capture-a_1/no-a	**if** capture-a(a',id) **is received and not** tried **and** id > a'	**send** no-a(id, a')
16. i		
17. capture-a_2/no-a	**if get** capture-a(a', id) **from** capture-a_message_queue **and not** tried **and** id > a'	**send** no-a(id, a')
18. capture-a_2/yes-a	**if get** capture(a', id) **from** capture-a_message_queue **and** (tried **or** a' > id)	**send** yes-a(id, a')
19. undo-a	**if** undo-a(a',id) **is received**	

20. commit-a/conf-a	**if** commit-a(a', id) **is received**	decrease_priority(a');
		if a□ conflict then tried := false;

Initialization
M1 is in the Inactive-a state, M2 is in the Input-a state and M3 is in the Execution-a state; captured_all_a:= false; captured_all_p:= false; tried:= false; action_priority_queue := sorted all actions of the system; Capture-a_message_queue := empty.

(C) The behavior of a P-process
It is the same as the behavior of a P-process defined in Algorithm 1 except that transition 3 is deleted (see Section 3.1.2).

3.3.2.3. Discussion

The proof of correctness of Algorithm 4 is given in [14]. Algorithm 4 satisfies su-fairness. In the algorithm, *capture-p* messages concerning only a single action can be sent out at any given time. Actions have different priorities. A *capture-p* message concerning an action can be sent out only if all actions having higher priorities have false guards. Priorities of actions change dynamically. The priority of an action is decreased after the action is executed.

4. Discussion and Conclusions

In this paper, we used labeled transition systems to model distributed systems with fairness. The so-called w-fairness, s-fairness, u-fairness and su-fairness are defined and four N-party synchronization algorithms are presented. Algorithms 1 and 2 satisfy w-fairness and u-fairness, respectively. Algorithm 3 respects w-fairness and u-fairness. Instead of directly implementing s-fairness, Algorithm 4 is designed to satisfy su-fairness which is stronger than s-fairness.

Actually, Algorithm 1 is neither new nor optimized. The idea of capturing processes one by one in a pre-defined order was used by Ramesh [11] as well as Kumar [8]. What is new about Algorithm 1 is that it is more abstract and simple, which makes the reasoning about fairness easier. In fact, both Ramesh's and Kumar's algorithms can be seen as optimizations of our Algorithm 1 (see [14] for details). Algorithm 2 is new and satisfies u-fairness. It forces process synchronize at their idle states which is implicitly required by u-fairness. Both Ramesh's and Kumar's algorithms satisfy w-fairness, but not u-fairness. Since the two algorithms can be seen as two optimizations of our Algorithm 1, we conclude that, we can modify these two algorithms to satisfy u-fairness, as we did for Algorithm 1 to obtain Algorithm 2.

Algorithm 3 is new and shows another scheme for implementing N-party synchronization. Algorithm 4 is constructed based on the scheme of Algorithm 3. The idea of synchronizing A-processes is also used in the algorithm of [5]. In the algorithm, the system first finds actions which are enabled, and then solves the conflicts among these actions. Algorithm 4 satisfies su-fairness and s-fairness by using the key idea of introducing priority among actions and forcing processes to synchronize on their idle-states.

The algorithms given in this paper are abstract in the sense they are not optimized. The abstraction simplifies the construction of the algorithms and reasoning of their correctness. We show, from the design of the algorithms, that the stronger the fairness expected form an algorithm, the more synchronization (global information) is required.

Reference

1. R. C. Attie, I. R. Forman and E. Levy, *On Fairness as an abstraction for the design of distributed systems,* Proc. of The 10th International Conference on Distributed Computing Systems, Paris, France, 1990.
2. R.J.R. Back and R. Kurki-Suonio, *Serializability in distributed systems with hand-shaking*, in Proc. 15th ICALP, Tampere, LNCS 317, pages 52-66, Springer-Verlag, Juillly 1988.
3. A. Charlesworth, *The Multi-way Rendezvous*, ACM Tran. on Programming Languages and Systems, Vol. 9, No. 2, July 1987, pp. 350-366.
4. N. Francez, *Fairness,* Springer-Verag New York, 1986.
5. Q. Gao and G. v. Bochmann, *Distributed Implementation of LOTOS Multiple-Rendezvous,* participant proceeding of The 9th International Symposium of Protocol Specification, Testing, and Verification, Enschede, The Netherlands, 1989.
6. ISO, *LOTOS: a formal description technique,* IS8807, 1989.
7. R. Kuiper and W. P. d. Roever, *Fairness assumptions for CSP in a temporal logic framework,* Proc. of TC.2 Working Conference on the Formal Description of Programming Concepts, Garmisch Partenkirchen, North Holland, 1983.
8. D. Kumar, *An implementation of N-way Synchronization Using Tokens,* Proc. of The 10th International Conference on Distributed Computing Systems, Paris France, 1990.

9. J. Parrow, *Fairness properties in process algebra,* (Ph.D. thesis) Dept. of computer Systems, Uppsala University, Uppsala, Sweden, 1985.
10. A. Pnueli, *The Temporal Semantics of Concurrent Programs,* LNCS 70, Springer-Verlag, 1979, pp. 1-20.
11. S. Ramesh, *A New and Efficient Implementation of Multiprocesses Synchronization,* Proc. of PARLE, Eindhoven, 1987.
12. Yih-Kuen Tsay, Rajive L.Bagrodia, "Some Impossibility Results in Interprocess Synchronization", Technical report CSD-890059, Computer Science Department, UCLA, 1989
13. C. Wu and G. v. Bochmann, *Fairness in LOTOS,* FORTE'91, Sydney, Nov. 19 - 21, 1991.
14. C. Wu, G. V. Bochmann, and Minyu Yao, *Fairness of N-party Synchronization and Its Implementation in a Distributed Environment,* No. 797, Dept. I.R.O., Universite de Montreal, Nov. 1991