Distributed termination detection: General model and algorithms

Jerzy Brzezinski, Jean-Michel Hélary, Michel Raynal

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Distributed Termination Detection : General model and Algorithms

Jerszy Brzezinski, Jean-Michel Hélary and Michel Raynal

N° 1964

March 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

ISSN 0249-6399

Distributed Termination Detection : General model and Algorithms

Jerszy Brzezinski*, Jean-Michel Hélary and Michel Raynal**

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet Algorithmes Distribués et Protocoles

Rapport de recherche n°1964 — March 1993 — 21 pages

Abstract: Termination detection constitutes one of the basic problems of distributed computing and many distributed algorithms have been proposed to solve it. These algorithms differ in the way they ensure consistency of the detection and in the assumptions they do concerning behaviour of channels (FIFO or not, bounded delay or asynchronous, etc). But all these algorithms consider a very simple model for underlying application programs : for processes of such programs non-deterministic constructs are allowed but each receive statement (request) concerns only one message at a time. In this paper a more realistic and very general model of distributed computing is first presented. This model allows a request (receive statement) to be atomic on several messages and to obey *AND/OR/AND-OR/k out of n/etc* request types. These request types are abstracted by the notion of an activation condition. Within this framework two definitions of termination are proposed and discussed. Then, accordingly, two distributed algorithms to detect these terminations are presented and evaluated ; they differ in the information they use and in the time they need to claim termination.

(Résumé : tsvp)

This work has been partly supported by KBN Grant 335209102 (KBN 2), by INRIA (visit of J.Brzezinski to IRISA) and by the Commission of the European Communities under ESPRIT Program Basic Research Project 6360 (BROADCAST)

This article will appear in the proceedings of the 13th IEEE Int. Conf. on Dist. Comp. Systems, Pittsburgh, USA, 25-28 May 1993

*Institute of Computing Science, Technical University of Poznan, 60-965 Poznan, POLAND, brzezins@plpotu51.bitnet

**IRISA, {helary}{raynal}@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

Détection répartie de la terminaison : modèle et algorithmes généraux

Résumé : La détection de la terminaison constitue l'un des problèmes de base du calcul réparti et de nombreux algortihmes répartis ont été proposés pour le résoudre. Ces algorithmes se distinguent par la manière d'assurer la cohérence de la détection et par les hypothèses de comportement des canaux. Mais ils considèrent tous un modèle très simple de calcul sous-jacent : chaque instruction de réception n'autorise qu'un seul message à la fois. Dans cet article, un modèle plus réaliste et très général est présenté : il autorise des instructions de réception atomiques sur plusieurs messages, avec des types *ET/OU/ET-OU/k parmi n/etc*. Le concept de condition d'activation constitue une abstraction de ces divers types. Dans ce cadre, deux définitions de la terminaison sont proposées et discutées, auxquelles correspondent deux algorithmes répartis, évalés et démontrés : ils se distinguent par l'information utilisée et par le délai de détection

1 Introduction

Since it was brought into prominence in 1980 by Francez ([4]) and by Dijkstra and Scholten ([3]), the distributed termination problem constitutes a classical problem of distributed control, due to its practical and theoretical importance. Considering a distributed execution of an application program (usually called the underlying computation) the aim is to construct a superimposed control program, that will detect the termination of the underlying computation. Such a detection is not trivial as no process has complete knowledge of the global state of the computation, and as global time or common memory do not exist in a distributed setting.

According to Mattern : "A distributed computation is considered globally terminated if every process is locally terminated and no messages are in transit. Locally terminated can be understood to be a state in which a process has finished its computation and will not restart unless it receives a message" (see Section 2 of Introduction of [7]). This definition is implicitly based on the following model for distributed computations :

- an active process can execute statements modifying its internal state, send messages or become spontaneously passive (i.e. locally terminated),
- a passive process can only receive messages (one at a time) and then instantaneously it becomes active.

In this context termination means that all processes are passive and all messages sent have been received and interpreted(consumed) by their destination processes. Many distributed algorithms have been designed to detect such a termination. In excellent paper [7] Mattern presents several efficient algorithms to detect termination within this model and provides a long list of references concerning the subject.

The major advantage of the previous model of distributed computations (and consequently the associated definition of termination) lies in its simplicity. But it is not general enough to take into account some realistic distributed computations. Let us consider, for example, the following situation in which process P_i atomically requests either two messages from P_a and P_b or one message from P_c. It is important to note that messages from P_a and P_b must both be consumed indivisibly if any ; otherwise, what can be done if a message from P_a is consumed, a message from P_c arrives and P_b never sends a message to P_i ?

In this paper a very general model for distributed computations is first presented (Section 2). This model allows for the possibilities of *AND, OR, AND-OR, k out of n* receive statements (requests) and it takes into account unspecified receptions (i.e. pending messages) which have never been required and thus consumed by their destination processes. Of course the presence of unspecified receptions certainly means the underlying program is in some sense incorrect, but realistic termination detection algorithms have to cope with correct as well as with incorrect programs. As the classical definition of termination (all the messages have been consumed) does not cover the above mentionned cases, the Section 2 introduces the notions of *static* and *dynamic* termination well-suited to the general model. Static termination requires that all channels are empty. In dynamic termination it is allowed that some (not required) messages are still in transit.

Then, two general distributed algorithms for detection of static and dynamic termination are presented (in Sections 3 and 4 respectively). The first algorithm (Section 3) assumes that each application message is acknowledged. This results in a "late" detection as termination is detected when all messages have arrived at their destinations (but some of them need not be consumed). The second algorithm (Section 4) applies a global counting technique in which each control message carries a vector of counters ; this additionnal information enables an "early" detection as termination can be detected even if some application messages have not yet arrived.

2 Model and problem formulation

2.1 Model of distributed computations

2.1.1 Communication model

A distributed application program (whose execution is traditionally called underlying computation) is composed of a finite set *P* of processes $P_i, i = 1, \ldots, n$, interconnected by unidirectional transmission channels ; the channel C_{ij} links the sender P_i to the receiver P_j. Processes communicate only by exchanging messages through channels ; there is neither common memory nor global clock.

Communication is asynchronous in the following sense :

- a sender sends a message to a channel (which then has responsability for its delivery) and immediately continues its execution ;
- channels do not necessarily obey the FIFO rule, but they are reliable (no loss, no corruption, no duplication, no spurious messages) ;
- a channel transfers (carries) a message till its destination process (receiver) and puts it in its local buffer : the message has then *arrived* ; the arrived message can then be consumed provided that its receiver has been activated, i.e. when the request of receiver has been fulfilled (see Section 2.2 for request models).
- the transfer delay (time elapsed between sending and arrival of a message) is finite but unpredictable.

As we can see, a message that is never consumed remains indefinitely in a buffer at the destination process : it reveals an *unspecified reception* (forever pending message).

2.1.2 Process model

At any time a process is either *active* or *passive*. An active process can execute internal computations (not involving messages), send messages and become passive by requiring some messages in order to continue its execution. This requirement is expressed by an *activation condition* (see Section 2.2) defined over the set DS_i of processes from which passive process P_i is expecting messages. The set DS_i associated with passive process P_i is called *dependent set* of P_i. A *passive* process can only become active when its activation condition is fulfilled. If such an activation is realized as soon as the activation condition is fulfilled (i.e. without any additional delay with respect to the activation condition fulfilment), we speak of *instantaneous* activation.

A passive process that has terminated its computation, executing for example an *end* or *stop* statement, is said to be *individually terminated* ; its dependent set is empty and therefore it can never be activated.

2.2 Request models

Formulation of activation conditions strictly depends on the request model considered. As it is important for further considerations, the main request models and a general predicate *fulfilled* are introduced. In each case, arrived messages that provoke an activation are then consumed after activation of their receivers.

2.2.1 AND model

In this model a passive process P_i can be activated only after a message from each process P_j belonging to DS_i has arrived. It models a receive statement that is atomic on several messages.

2.2.2 OR model

In *OR* model, a passive process P_i can be activated when a message from any process P_j belonging to DS_i has arrived. It models classical non-deterministic receive constructs.

2.2.3 OR-AND model

For *OR-AND* model, the requirement of a passive process P_i is defined by a set R_i of sets DS_i^1 , DS_i^2, ..., $DS_i^{q_i}$, such that for all $r, 1 \leq r \leq q_i, DS_i^r \subseteq P$. The dependent set of P_i is : $DS_i = DS_i^1 \cup DS_i^2 \cup \ldots \cup DS_i^{q_i}$. We mean here that process P_i waits for messages from *all* processes belonging to DS_i^1, or for messages from *all* processes belonging to DS_i^2, *or ..., or* for messages from *all* processes belonging to $DS_i^{q_i}$. As an example, suppose P_i waits for messages from : P_a *or* $(P_b$ *and* $(P_c$ *or* $(P_d$ *and* $P_e)))$. In disjunctive form this gives : P_i waits for P_a *or* $(P_b$ *and* $P_c)$ *or* $(P_b$ *and* P_d *and* $P_e)$. In this case $DS_i^1 = \{P_a\}, DS_i^2 = \{P_b, P_c\}$ and $DS_i^3 = \{P_b, P_d, P_e\}$. Let us note, that if messages from P_c and P_d have arrived, and then a message from P_b arrives, the activation condition of P_i is fulfilled. This activation provokes the consumption of the messages from P_b and P_c.

2.2.4 Basic *k out of n* model

In this model the requirement of a passive process P_i is defined by the set DS_i and an integer $k_i, 1 \leq k_i \leq |DS_i| = n_i$. Process P_i can be activated when messages from k_i distinct processes belonging to DS_i have arrived.

2.2.5 Disjunctive *k out of n* model

A more general request model can be introduced including disjunctions of *k out of n* requests. The requirement of a passive process P_i is defined by a set R_i, as previously (Section 2.2.3), and by a set of integers $K_i = \{k_i^1, k_i^2, \ldots, k_i^{q_i}\}$ with $1 \leq k_i^r \leq |DS_i^r| = n_i^r$ for all $r, 1 \leq r \leq q_i$. The dependent set of P_i is : $DS_i = DS_i^1 \cup DS_i^2 \cup \ldots \cup DS_i^{q_i}$. A process P_i is activated when :

messages from k_i^1 processes composing DS_i^1 have arrived *or*
messages from k_i^2 processes composing DS_i^2 have arrived *or*
... *or*
messages from $k_i^{q_i}$ processes composing $DS_i^{q_i}$ have arrived.

Let us note, that if for all $r, 1 \leq r \leq q_i, k_i^r = n_i^r = |DS_i^r|$ this model reduces to the *OR-AND* model. On the other hand, when $q_i = 1$ and $k_i^1 \leq |DS_i^1|$, then we have the basic *k out of n* model, with $k = k_i^1$ and $n = n_i^1$. The *AND* model is deduced when $q_i = 1, DS_i^1 = DS_i$ and $k_i^1 = n_i^1 = |DS_i|$. The simple *OR* model is obtained when $q_i \geq 1$ and for all $r, 1 \leq r \leq q_i, |DS_i^r| = 1$.

2.2.6 Predicate *fulfilled*

Finally, in order to abstract the activation condition of a passive process P_i, the following predicate $fulfilled_i(A)$ is introduced, where *A* is a subset of *P*. Predicate $fulfilled_i(A)$ is *true* if and only if messages arrived (and not yet consumed) from all processes belonging to set *A* are sufficient to activate process P_i. Of course the following monotonicity property is valid : if $X \subseteq Y$ and $fulfilled_i(X)$ is *true*, then $fulfilled_i(Y)$ is also *true* ; moreover $fulfilled_i(\phi)$ is *false*.

If we consider the previous disjunctive *k out of n* model the predicate is expressed as follows. Let P_i be a passive process whose requirements are defined by the sets R_i and K_i. Then we get the following definition :

$$fulfilled_i(A) \equiv \exists r : 1 \leq r \leq q_i : |DS_i^r \cap A| \geq k_i^r$$

Similar definition can be obtained for the other models.

2.3 Termination definitions

2.3.1 Notation

The following notations are introduced to formally define terminations of distributed computations.

- $passive_i$: *true* iff P_i is passive.
- *empty(j,i)*: *true* iff all messages sent by P_j to P_i have arrived at P_i ; the messages not yet consumed by P_i are in its local buffer.
- $arr_i(j)$: *true* iff a message from P_j to P_i has arrived and has not yet been consumed by P_i.
- ARR_i=$\{$processes P_j such that $arr_i(j)\}$.
- NE_i=$\{$processes P_j such that $\neg$ *empty(j,i)*$\}$.

2.3.2 Dynamic termination (DT)

The set *P* of processes is said to be *dynamically terminated* at some time if and only if the predicate *Dterm* is true at this moment, where:

$$Dterm \equiv \forall P_i \in P : \ passive_i \wedge \neg fulfilled_i(ARR_i \cup NE_i)$$

This notion of termination means that no more activity is possible from processes, though messages of the underlying computation can still be in transit (represented by possibly non empty sets NE_i in the predicate). This definition is interesting for "early" detection of termination as it allows to conclude a computation is terminated even if some of its messages have not yet arrived. This definition is mainly focused on the real ($\neg passive_i$) or potential ($fulfilled_i(ARR_i \cup NE_i)$) activity of processes. Note that the condition of potential activity is here considered instead of checking whether all channels are empty.

Of course, one can show that, once true, the predicate *Dterm* remains true. Thus, dynamic termination is a stable property.

2.3.3 Static termination (ST)

The set *P* of processes is said to be *statically terminated* at some time if and only if the following predicate *Sterm* is true at this moment:

$$Sterm \equiv \forall P_i \in P : passive_i \wedge (NE_i = \phi) \wedge \neg fulfilled_i(ARR_i)$$

For this predicate to be true, channels must be empty and processes cannot be activated. Thus, this definition is focused on the state of both channels and processes. When compared to *Dterm*, the predicate *Sterm* corresponds to "late" detection as, additionally, channels must be empty. Following [2] let $p \mapsto q$ denote the *leads-to* relation over predicates *p* and *q*, meaning that once *p* is true, *q* is true or will eventually be true.

Theorem 1

$$Dterm \mapsto Sterm$$

Proof (outline)
When *Dterm* is true, at time *t*, all processes are passive but all channels are not necessarily empty. However all not yet arrived messages are taken into account and their arrivals are anticipated evaluating predicates $fulfilled_i(ARR_i \cup NE_i)$.

Thus arrival of all these messages, in finite although unpredictable time, does not change the value of predicate ; but then at $t' \geq t$ when all channels are empty we have the value of ARR'_i (ARR_i at *t'*) equal to the value of $ARR_i \cup NE_i$ at *t*. Thus *Sterm* becomes true at *t'*. □

It should be stressed that the delay between the time moments when*Dterm* and, accordingly, *Sterm* become *true*, is finite but unpredictable.

Choice of one definition of termination or the other depends essentially on the user. Among others interesting benefits, the occurrence of *Dterm* allows to consider as definitive (and consequently use as such) the results of the underlying computation whereas some of its messages are still in transit ! The following section shows that the classical definition of termination is in fact a special case of the static definition.

2.3.4 Instantaneaous activation and classical model

Recall that instantaneous activation means that processes are activated immediately when their activation conditions become true (Section 2.1.2).

Theorem 2 *Considering instantaneous activation, the set P of processes is statically terminated if and only if the following predicate is true :*

$$\forall P_i \in P \ : \ passive_i \wedge (NE_i = \phi)$$

Proof
If processes are instantaneously activated, they become active immediately when $fulfilled_i(ARR_i)$ becomes *true* and consequently $passive_i$ is *true* only when $\neg fulfilled_i(ARR_i)$. So we get

$$passive_i \wedge \neg fulfilled_i(ARR_i) = passive_i \qquad \Box$$

In the classical model of distributed computations each receive statement concerns only one message at a time and unspecified receptions are not considered. Note, that restricting considerations to this model and assuming moreover instantaneous activation, static termination reduces to the classical one.

2.4 Termination detection problem

Given a distributed program consisting of a fixed set *P* of processes which cooperate realizing a distributed computation, the problem is to detect its static or dynamic termination occurrence, i.e. a system state in which the corresponding predicate is true. To achieve this goal a termination detection algorithm is used whose execution constitutes a superimposed control computation. On the one hand, this detection computation must not affect the behaviour of the underlying computation and, on the other hand, it should detect (static or dynamic) termination as soon as possible because results of the application program are ready since termination occurrence and no useful performance can be expected further. Thus, any delay in the detection means degradation of possible performance.

As we already mentioned, delay between static and dynamic termination occurrences is finite but unpredictable. Thus in the efficiency context detection of dynamic termination is much more attractive because it allows (at least potentially) to avoid this unpredictable delay. It can be especially important in real-time

applications.

Classically, correctness of termination detection algorithms is expressed by the two following properties :

- Safety property (consistency) :
  If the detection algorithm claims termination then the underlying computation has terminated.

- Liveness property (progress) :
  If the detection algorithm is executing when the underlying computation terminates, then it will claim termination in finite time.

Moreover as far as efficiency is concerned (whatever detection is done : static or dynamic), the algorithm shoud be characterized by a detection delay as short as possible and by a smallest possible overhead (in terms of communication and storage complexities).

The two following sections develop two distributed algorithms to detect termination of distributed computations expressed in the context of the general model just introduced. The first algorithm (Section 3) is concerns detection of static termination : it computes the occurrence of predicate *Sterm*. The second algorithm (Section 4) is concerns dynamic termination, characterized by predicate *Dterm*.

3 Detection of static termination : the STD algorithm

3.1 Informal description of the STD algorithm

A *control* process C_i, called *controller*, is associated with each application process P_i. Its role is, on the one hand, to observe the behaviour of P_i and, on the other hand, to cooperate with other controllers C_j in order to consistently detect occurrence of the predicate *Sterm*. (In general, controllers need not be separated as special processes since their tasks can be incorporated into application processes using the superimposition rules described e.g. in [2]. Thus, the separation of controllers is merely a matter of interpretation).

3.1.1 State variables

Each process P_i is endowed with a state variable $state_i$ whose value, readable by C_i, is *active* or *passive* ($passive_i$ is a shortcut for $state_i = passive$). As introduced in Section 2.1, each process is equipped with input buffers where messages arrived and not yet consumed are stored. Let us recall that ARR_i is the set of senders of messages arrived to P_i but not yet consumed by P_i. Controller C_i can atomically read the value of ARR_i.

For the predicate *Sterm* to be true, all the messages sent have to be arrived. A way to acquire this knowledge is to use an acknowledgement mechanism. If the underlying communication system does not provide such a mechanism, each controller C_i first acknowledges each message arrived and then manages a counter $notack_i$ counting the number of messages sent by P_i and not yet acknowledged.

3.1.2 The detection strategy

A sequence of waves.
In order to detect static termination, a controller, say C_α, initiates a detection by sending to all controllers (including itself) a control message *query*. All controllers C_i will answer with a boolean valued message $reply(ld_i)$. Then C_α combines all these answers computing $td := \bigwedge_{1 \leq i \leq n} ld_i$. If *td* is *true*, C_α claims termination ; in the other case it anew sends *query* messages. The basic sequence of one sending of *query* messages followed by the reception of associated *reply* messages is usually called a *wave*. Let us remark waves are sequential. (The interested reader will find a theory of waves in [6], and basic implementations of waves in [5,8,9]).

The basic test.
The core of the algorithm is the way a controller C_i computes the value ld_i sent back as answer for a *query* message. To ensure safety, the values $ld_1, \ldots, ld_n$ must be such that :

$$ld_1 \wedge \ldots \wedge ld_n \Rightarrow Sterm$$

thus :

$$\bigwedge_{1 \leq i \leq n} ld_i \Rightarrow \forall P_i \in P : \; passive_i \wedge NE_i = \phi \wedge \neg fulfilled_i(ARR_i)$$

First of all, a controller C_i delays its answer as long as the following (locally evaluable) predicate is false :

$$passive_i \wedge notack_i = 0 \wedge \neg fulfilled_i(ARR_i)$$

When this predicate is false, the static termination cannot be guaranteed (note that the second factor of this predicate concerns output channels, whereas the corresponding part of *Sterm* concerns input channels).

Second, in order the values reported by a wave be globally correct they must not miss activity of processes "in the back" of the wave (controllers receive *query* messages and send back *reply* messages asynchronously). Therefore each controller C_i manages a boolean indicator cp_i (initialized to *true* iff P_i is initially *passive*) in the following way (similar to [1]) :

- when P_i becomes *active* (its activation condition is then *true*) cp_i is set to *false*.
- when C_i sends a *reply* message to C_α first it adds the current value of cp_i to this message, and then assigns to cp_i the value *true*.

Thus, reply message carrying value *true* from C_i to C_α means that : P_i has been continuously passive since the previous wave, and messages arrived and not yet consumed are not sufficient to activate P_i, and all output channels of P_i are empty. The proof will show the consistency of this strategy.

3.2 Formal description of the STD algorithm

The local variables used by each controller C_i have been introduced in the previous Section. The behavior of each C_i is described by the following statements *S1* to *S5*. All these statements are atomically executed except, of course, the *wait* instruction. Additionally, the controller C_α initiating a detection executes the statement *S6* (in which *receive* instruction shows interruptible points). By *a message* we mean here any message of the underlying computation ;*queries* and *replies* are called *control messages*.

$S1$: **when** P_i **sends** $a\ message$ **to** P_j
$notack_i := notack_i + 1$

$S2$: **when** $a\ message$ **from** P_j **arrives to** P_i
send ack **to** C_j

$S3$: **when** C_i **receives** ack **from** C_j
$notack_i := notack_i - 1$

$S4$: **when** P_i **becomes** $active$
$cp_i := false$
%$a\ passive\ process\ can\ only\ become\ active\ when\ its\ activation\ condition\ is\ true$; $this\ activation\ is\ under\ the\ control\ of\ the\ underlying\ operating\ system,\ and\ the\ termination\ detection\ algorithm\ only\ observes\ it$%

$S5$: **when** C_i **receives** $query$ **from** C_α
wait until $(passive_i \wedge notack_i = 0 \wedge \neg fulfilled_i(ARR_i))$;
$ld_i := cp_i$;
$cp_i := true$;
send $reply\ (ld_i)$ **to** C_α

$S6$: **when** $controller\ C_\alpha$ **decides to detect static termination**
repeat send $query$ **to all** C_i;
receive $reply\ (ld_i)$ **from all** C_i;
$td := \bigwedge_{1 \le i \le n} ld_i$;
until td;
$claim\ static\ termination$

3.3 Correctness of the algorithm

3.3.1 Notations

Considering execution of STD algorithm, let us denote by t_i^k the time moment -for a global observer- at the end of the *k*-th wave visit at controller C_i, i.e. when C_i sends *reply*. Let t_α^k be the end of the *k*-th wave (when C_α computes *td*). So, we have $t_i^k < t_\alpha^k < t_\alpha^{k+1}$.

Moreover, for any entity *X* (predicate, variable, etc) and any time *t*, *X*[*t*] will denote the value of *X* at time *t*. Finally, in order to make the proof easier to follow, C_α is considered as an extra control process (not associated with any P_i).

3.3.2 Liveness

If the STD algorithm is executing when the underlying computation statically terminates, then it will stop in finite time (or equivalently *td* will be equal to *true*).

Proof
If the underlying computation is statically terminated at time *t*, then since that time : all processes are continuously passive and their requirements are not fulfilled ($fulfilled_i(ARR_i) = false, \forall i$). Moreover, all channels are empty of underlying messages and, therefore, all the variables $notack_i$ will decrease to value 0 in finite time.

Thus, the current wave (wave at time *t*) and all the subsequent ones will not be delayed by the controllers (in the **wait until** statement). The first next wave, let it be the *k*-th, will set all local flags cp_i to *true*. The wave $k+1$ will set to *true* all the local variables ld_i (if not already done) and hence in finite time the variable *td* will be given the value *true*. Then the STD algorithm terminates. □

3.3.3 Safety

If the STD algorithm claims termination (*td* has then the value *true*), then the underlying computation is statically terminated.

proof
Let us consider two consecutive waves *k* and *k+1* and suppose that the STD algorithm claims termination at the end of the *k+1*-th wave, i.e. at t_α^{k+1} (at that time we have *td=true*). As waves are sequential there exists a time moment σ^k such that, for any *i* :

$$t_i^k \leq t_\alpha^k \leq \sigma^k \leq t_i^{k+1} \leq t_\alpha^{k+1}$$

We will prove that at time $t = \sigma^k$ the underlying computation is statically terminated, i.e., $Sterm[\sigma^k]$ is true. In other words, for all processes P_i of the underlying computation the following conditions hold at σ^k :

$$C1 : passive_i$$
$$C2 : NE_i = \phi$$
$$C3 : \neg fulfilled_i(ARR_i)$$

i) proof of *C1*

By construction STD algorithm terminates at t_α^{k+1} with *td=true*, only if each process P_i has been continuously passive between t_i^k and t_i^{k+1}. But as for each P_i we have $t_i^k \leq \sigma^k \leq t_i^{k+1}$: we conclude all processes were passive at σ^k. So *C1* is verified.

ii) proof of *C2*

By construction each wave (and therefore also the *k*-th one) is kept at each controller C_i until all underlying messages sent by P_i have been acknowledged ($notack_i = 0$). On an other hand each process cannot send messages after the *k*-th wave visit as it is continuously passive between the *k*-th and the *k+1*-th wave visits, i.e. between t_i^k and t_i^{k+1}. So we conclude all channels are empty at time σ^k.

iii) proof of *C3*

Again, by construction a wave is kept at controller C_i until $\neg fulfilled_i(ARR_i)$ is true. Thus, as the STD algorithm terminates at t_α^{k+1}, $\neg fulfilled_i(ARR_i)[t_i^{k+1}]$ was true for each controller C_i. As each process P_i has been continuously passive between *k* and *k+1* wave visits, it consumed no messages in interval t_i^k to t_i^{k+1}, and therefore the set ARR_i could only increase during this interval :

$$ARR_i[\sigma^k] \subseteq ARR_i[t_i^{k+1}]$$

As $\neg fulfilled_i(ARR_i)[t_i^{k+1}]$ is true we conclude $\neg fulfilled_i(ARR_i)[\sigma^k]$ (see Section 2.2.6 for the monotomicity property attached to the definition of this predicate). That proves *C3*.

Thus, safety property has been proven. □

3.4 Performance analysis

The efficiency of STD algorithm depends on the implementation of waves (see [5,9] for canonical implementations based on rings and on spanning trees). As in

other detection algorithms we assume C_α is connected to all controllers.

Two waves are in general necessary to detect static termination. A wave needs two types of messages : *n* queries and *n* replies carrying one bit. Thus, neglecting acknowledgement messages (whose number is the same as the number of underlying messages), *4n* control messages of two distinct types and carrying at most one bit each are used to detect termination once it occurred. Let us note that if waves are supported by a ring this complexity can easily be reduced to *2n*. Finally the detection delay is equal to the duration of two sequential wave executions.

3.5 Initiation of the STD algorithm

The role of C_α can in fact be played either by only one (statically or dynamically defined) controller or by each controller. (In this latter case each control message carries the identity of the associated launching controller and each controller multiplexes its behaviour on execution of all these STD algorithms). Selection of one or several detection executions does not depend on the algorithm but on its users.

4 Detection of dynamic termination : the DTD algorithm

4.1 Informal description of the DTD algorithm

As shown by predicate *Dterm* dynamic termination can occur before all messages of the underlying computation have arrived, and consequently this termination can be detected earlier than static one.

To detect occurrence of *Dterm* a wave mechanism very similar to the previous one (Sections 3.1 and 3.2) is used by DTD algorithm. C_α denotes the controller sequentially launching the waves. Local variables cp_i have the same meaning as before. Now each controller C_i is endowed with the following additional control variables denoted s_i and r_i, two vectors counting messages respectively sent to and received from each other process P_j :

$$s_i[j] \;=\; number\ of\ messages\ sent\ by\ P_i\ to\ P_j$$

$$r_i[j] \;=\; number\ of\ messages\ received\ by\ P_i\ from\ P_j$$

Moreover let *S* denote an $n \times n$ matrix of counters used by C_α ; its columns are carried by *query* messages ; the entry *S*[*i,j*] represents C_α's knowledge about the number of messages sent by P_i to P_j.

First, C_α sends to each C_i a message *query* containing the vector $(S[1,i],\ldots,S[n,i])$ denoted by $S[.,i]$. Upon receiving such a message C_i can compute the set ANE_i of its non-empty channels (that is, of course, an approximate knowledge but sufficient to ensure correctness, see Section 4.3). Then C_i computes its local answer ld_i that is true if and only if P_i has been continuously passive since the previous wave and its requirements cannot be fulfilled by all the messages arrived and not yet consumed (ARR_i) and all the messages potentially in its input channels (ANE_i). Afterwards C_i sends to C_α a *reply* message carrying the values of ld_i and of its vector s_i (these last values will be used by C_α to update row *S*[*i,*.] and thus to gain more accurate knowledge). If $\bigwedge_{1\leq i\leq n} ld_i$ evaluates to *true*, C_α claims dynamic termination of the underlying computation. Otherwise, C_α launches a new wave sending again *query* messages.

As we can see, when compared to previous STD algorithm, variables $notack_i$ and acknowledgements are not used in the DTD algorithm. They have been replaced by more individualized vector variables s_i and r_i. These vectors allow respectively to update C_α's (approximate) global knowledge about messages sent by each P_i to each P_j, and to get an (approximate) knowledge of the set of non-empty input channels. Finally, let us note that, as the DTD algorithm concerns "early" detection, waves are not delayed by controllers.

4.2 Formal description

All controllers C_i execute statements *S1* to *S4*. Only the initiator C_α executes *S5*. Local variables s_i, r_i and *S* are initialized to 0.

$S1$: **when** P_i **sends** $a\ message$ **to** P_j
$\quad s_i[j] := s_i[j] + 1$

$S2$: **when** $a\ message$ **from** P_j **arrives to** P_i

$r_i[j] := r_i[j] + 1$

$S3$: **when** P_i **becomes** $active$
$cp_i := false$

$S4$: **when** C_i **receives** $query(VC[1..n])$ **from** C_α
$\%VC[1..n] = S[1..n, i]$ is the i-th column of S %
$ANE_i := \{P_j : VC[j] > r_i[j]\}$;
$ld_i := cp_i \wedge \neg fulfilled_i(ARR_i \cup ANE_i)$;
$cp_i := (state_i = passive)$;
send $reply(ld_i, s_i)$ **to** C_α

$S5$: **when** $controller\ C_\alpha$ **decides to detect dynamic termination**
repeat for each C_i : **send** $query(S[1..n, i])$ **to** C_i;
% the i-th column of S is sent to C_i %
receive $reply(ld_i, s_i)$ **from all** C_i;
$\forall i \in [1..n]\ :\ S[i, .]\ := s_i$;
$td := \bigwedge_{1 \le i \le n} ld_i$;
until td;
$claim\ dynamic\ termination$

4.3 Correctness of the DTD algorithm

The notations used have been introduced in Section 3.3.1 .

4.3.1 Liveness

If the DTD algorithm is executing when the underlying computation dynamically terminates, then it will stop in finite time (and then *td* will be true).

Proof
As waves are not delayed by controllers, C_α will get the result of each wave in finite time.

If the underlying computation dynamically terminates at time *t* (*Dterm*[*t*] is true), then since this instant each process P_i is continuously passive and its requirements

cannot be fulfilled : $\neg fulfilled_i(ARR_i \cup NE_i)[t']$ for $t' \geq t$. Thus the wave launched after *t*, denoted here by *k*, sets all variables cp_i to *true*, and collects final values of $s_i[j]$ counters. Therefore at time moments $t_i^{k+1} > t$ we have $ANE_i[t_i^{k+1}] = NE_i[t_i^{k+1}]$ and hence $\neg fulfilled_i(ARR_i \cup NE_i)[t_i^{k+1}]$ implies $\neg fulfilled_i(ARR_i \cup ANE_i)[t_i^{k+1}]$. Thus *k+1*-th wave will set to *true* all ld_i flags ; and finally all variables ld_i collected will give the value *true* to *td* and the DTD algorithm terminates. □

4.3.2 Safety

If the DTD algorithm claims termination (it ends its execution with *td=true*), then the underlying computation has dynamically terminated (*Dterm* is then true).

Proof
As in Section 3.3.3 the last two waves are denoted by *k* and *k+1*, and there exists a time moment σ^k such that, for any *i* :

$$t_i^k \leq t_\alpha^k \leq \sigma_k \leq t_i^{k+1} \leq t_\alpha^{k+1}$$

To prove safety we will show that if the DTD algorithm terminates at t_α^{k+1} with *td=true*, then $Dterm[\sigma^k]$ holds, i.e. the following conditions hold at σ^k for all processes P_i :

$$\begin{aligned} C4 &: passive_i \\ C5 &: \neg fulfilled_i(ARR_i \cup NE_i) \end{aligned}$$

i) proof of *C4*.
The proof is identical to *C1*'s one (Section 3.3.3), so it is omitted.

ii) proof of *C5*.
By construction, the DTD algorithm terminates at t_α^{k+1} provided that $\neg fulfilled_i(ARR_i \cup ANE_i)[t_i^{k+1}]$ was true for each controller C_i.

As each process has been continuously passive between *k* and *k+1* wave visits we have for any *i* :

$$(X1) \ : \ ARR_i[\sigma^k] \subseteq ARR_i[t_i^{k+1}] \text{and } s_i[j][t_i^k] = s_i[j][\sigma^k]$$

The DTD algorithm evaluates the set $ANE_j[t_j^{k+1}]$ of non-empty channels according to the relation (for C_j when visited by *k+1* wave) :

$$s_i[j][t_i^k] \ > \ r_j[i][t_j^{k+1}] \text{ (as we have } s_i[j][t_i^k] = S[i,j][t_\alpha^k]).$$

Moreover, each counter is monotonic :

$$r_j[i][t_j^k] \leq r_j[i][\sigma^k] \leq r_j[i][t_j^{k+1}]$$

Thus, as for any *i* : $s_i[j][t_i^k] = s_i[j][\sigma^k]$, the difference between (real) value of $NE_j[\sigma^k]$ and (approximate) value of $ANE_j[t_j^{k+1}]$ can result only from arrival of some messages ; in this case however the appropriate senders will be included in $ARR_j[t_j^{k+1}]$. Thus taking into account *X1* we can conclude for any *i* :

$$(ARR_i \cup NE_i)[\sigma^k] \subseteq (ARR_i \cup ANE_i)[t_i^{k+1}]$$

Then, due to the monotonicity property of the predicate $fulfilled_i$ (see Section 2.2.6), we can deduce for each controller C_i that :

$$\neg fulfilled_i(ARR_i \cup ANE_i)[t_i^{k+1}] \Rightarrow \neg fulfilled_i(ARR_i \cup NE_i)[\sigma^k]$$

Condition *C5* is thus verified, and hence safety of the DTD algorithm is proved. □

4.3.3 Performance analysis

As in Section 3, the DTD algorithm needs two waves after dynamic termination to detect it. As no acknowledgements are necessary, its message complexity is equal to *4n* and is lower than STD algorithm. However, messages are composed of *n* monotonically increasing counters. As waves are sequential, *query* (respt. *reply*) messages between C_α and each C_i (respt. each C_i and C_α) are received and processed in their sending order ; this FIFO property can be used to carry only, relatively to each counter, the difference between its current value and the sum of differences already sent. That decreases the size of control messages.

The detection delay is 2 waves but it is shorter than the delay of the STD algorithm as acknowledgement are not used. This possible acceleration can be important in many applications, in particular time critical ones.

5 Conclusion

In the paper, a very general and realistic model of distributed computations has been introduced. This model allows requests to be atomic on several messages and to obey a *AND/OR/AND-OR/k out of n* request-type, among other . These requests have been abstracted by the notion of activation condition, formally described by the predicate *fulfilled*. Concerning "real" programs (sometimes incorrect !) the model takes into account unspecified receptions, i.e. messages sent and never consumed by their destination processes. These pending messages certainly show incorrect programs, but termination detection concerns all distributed computations, not only the correct ones.

Two definitions of termination have been proposed. *Static* termination means that all processes of the underlying computation are passive and all the messages they sent have arrived (some may remain not consumed if there are unspecified receptions). If each request is restricted to be on one message at a time and there is no unspecified reception, the definition of static termination becomes very close to the classical definition of termination. *Dynamic* termination means that all processes of underlying computations are passive and cannot be activated (but some of the messages they sent can still be in transit). This kind of termination is particularly interesting when results of the underlying computation must be known as soon as possible, as it allows "early" detection. Each kind of termination has been characterized by a simple predicate.

Finally two distributed algorithms have been designed, proven correct and analyzed. The first one, called STD algorithm, detects static termination, whereas the second one, called DTD algorithm, detects dynamic termination. They have been methodically designed ; with respect to liveness, they are based on the wave mechanism ; with respect to safety, they check boolean expressions whose shape is as close as possible as the one of the relevant (abstract) predicate. Albeit different from the classical distributed termination detection algorithms (at least in the predicate they compute), algorithms STD and DTD share with them the use of basic counting techniques. Finally, these two algorithms can easily be extended to report unspecified receptions if desired.

References

[1] K. M. Chandy, J. Misra, L. M. Haas, *Distributed deadlock detection.* ACM TOCS, vol.1,2, (1983), pp. 144-156.

[2] K. M. Chandy, J. Misra, *Parallel program design : a foundation.* Addison Wesley, (1988), 516 p.

[3] E. W. D. Dijkstra, C. S. Scholten, *Termination detection for diffusing computation.* Inf. Processing Letters, vol.13,1, (1980), pp. 1-4.

[4] N. Francez *Distributed termination.* ACM TOPLAS, vol.2,1, (1980), pp. 42-55.

[5] J. M. Hélary, M. Raynal, *Synchronization and control of distributed systems and programs.* Wiley & Sons, (1990), 125 p.

[6] J. M. Hélary, M. Raynal, *Distributed evaluation : a tool for constructing distributed detection program.* Proc. Int. Conf. on Theory of Computing and Systems, Springer-Verlag, LNCS 601, (Galil, Dolev, Rodeh Ed.), (1992), pp. 184-194.

[7] F. Mattern, *Algorithms for distributed termination detection.* Distributed Computing, vol.2,3, (1987), pp. 161-175.

[8] F. B. Schneider, L. Lamport, *Paradigms for distributed programs.* In Springer-Verlag, LNCS 190, (1985), pp. 431-480.

[9] G. Tel, *Topics in distributed computing.* Cambridge Int. Series on Parallel Proc., Cambridge University Press, (1991), 256 p.

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399