Distributed Termination Detection.

We address here the problem of detecting the termination of distributed computations [**1**]. We are given N processes $P_0, \ldots, P_{N-1}$ which are placed at the nodes of a given *undirected, connected* graph G. We identify each process with the node where it is located. The N processes perform a *main computation* and they exchange *messages* to neighbouring processes in the graph. The arcs of the graph represent communication channels. Each process (and the corresponding node) may be either
- *active*, that is, it still performs a part of the main computation (and in this case the process and the node are labelled by A), or
- *idle*, that is, it has completed the part of the main computation which has been assigned to it by the last message it has received (and in this case the process and the node are labelled by I).

We have that:
(1) Only active processes may send messages.
(2) A process may change from idle to active only on receipt of a message.
(3) A process may change from active to idle at any time (thus, we not assume any knowledge on the duration of the parts of the computations assigned to the processes).

At Point (2) we may replace 'may' by 'must' because of Point (3).

We say that the main computation has *terminated* iff all processes are idle.

We say that termination has been detected by a termination detection algorithm if it is the case that a process, say P_0, enters a fixed state iff the main computation has terminated. To know whether or not the remaining processes have terminated, process P_0 has the ability of sending and receiving messages to all processes in the graph, but this should be done by sending and receiving messages only to and from neighbouring processes. The ability of sending and receiving messages is given to every process in the graph G.

The messages devoted to termination detection are collectively called *signals.* Signals are: (i) either *tokens* or (ii) *repeats*. We will see below how tokens and repeats are used by the termination detection algorithm.

The termination detection algorithm we look for, is an algorithm which: (i) at each node sends or receives tokens or repeats, and (ii) at each node modifies the state of the process at that node.

Moreover, the termination detection algorithm should satisfy the following conditions:
(1) The modification of each process to incorporate termination detection should be independent of the definition of the process.
(2) The termination detection algorithm should not indefinitely delay the main computation.
(3) No new communication channels should be added among the processes.
(4) The termination detection algorithm should operate at each node on the basis of the information, tokens, and repeats available at that node only.

We assume that we have computed a spanning tree T of the given undirected graph G with process P_0 at its root. This can be done by the distributed algorithm we have indicated in Section 4.12 of [**?**].

The proposed termination detection algorithm works by making use of waves of tokens and repeats moving along frontiers of the spanning tree T. A *frontier* of a tree is a set F of nodes such that every root-to-leaf path has exactly one node in common with F.

Initially, a contracting *token wave* goes inwards from the leaves to the root, and if it reaches the root without detecting termination, then a new wave, called *repeat wave*, is generated. This new wave moves outwards from the root to the leaves. As soon as this repeat wave reaches the leaves, a new wave of tokens starts moving inwards again. Notice that in some branches of the spanning tree the token wave may be moving inwards while the repeat wave is still moving outwards along other branches.

The algorithm works by applying in a distributed way, as long as possible, the rules that we will indicate in the Figures 1 and 2 below. If the left-hand-side of a rule depicted in these figures holds at a node and the corresponding condition, if any, is true, then the rule fires and the right-hand-side of that rule is realized at that node, regardless of the rules which are applied in other nodes at a previous or a later time.

In the spanning tree T of G every node has exactly one parent node, except the root which has no parent. Every node has a (possibly empty) list of children. The list of children is empty iff the node is a leaf.

In Figure 1 and Figure 2 below we have adopted the following conventions. The status s of a node may be either *active* (A) or *idle* (I). The color of a node may be either *white* ($\square$) or *black* ($\blacksquare$). A token t may be either *white* ($\triangle$) or *black* ($\blacktriangle$). A repeat is denoted by $\bullet$.
Here are the two *rules $M1$ and $M2$ for the main computation* (see also Figure 1).

Rule $M1$ is for sending a message along an arc. An active process sends a message m to one of its neighbours, and assigns to it a part of the main computation which is encoded by that message. The process which sends the message m becomes black, and this color encodes the fact that the process which *receives* the message m, has begun and not yet completed the part of the main computation encoded by m.

Rule $M2$. At any time an active process may become idle regardless of its color. It becomes idle when it has completed the part of the main computation encoded by the last message it has received.

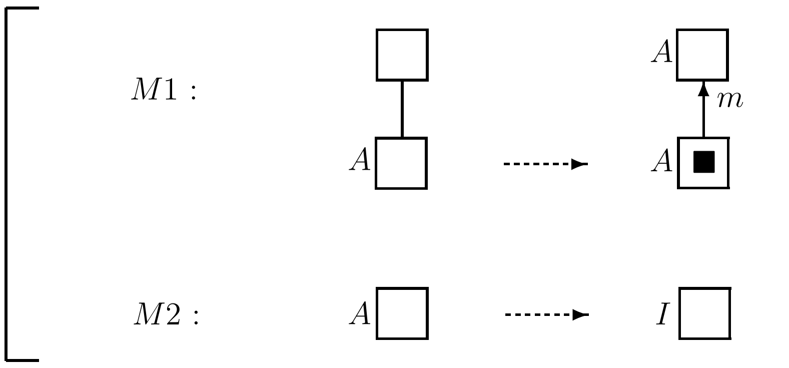

FIGURE 1. Rules of the main computation of the termination detection algorithm. The status s of a node may be either *active* (A) or *idle* (I). $\blacksquare$ denotes that the color of the node is *black*. m is a message of the main computation.

We do *not* assume that nodes remain active for a finite time only. Thus, if a node remains active for an infinite time, then rule $R2$ never fires at that node and the termination detection algorithm should never detect termination.
Here are the five *rules for the termination detection algorithm* (see also Figure 2).

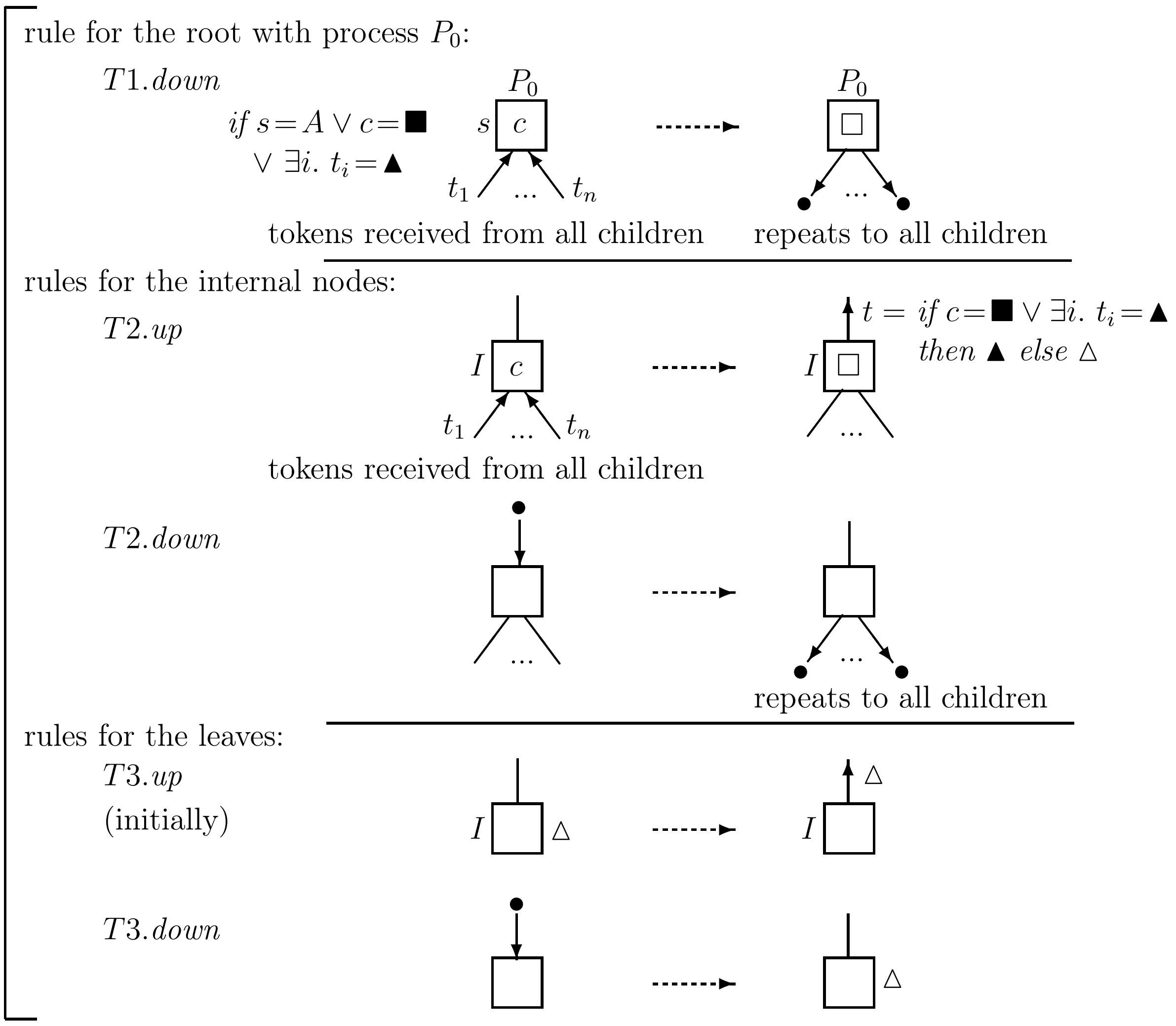
rule for the root with process P_0:
$T1.down$
P_0
if $s=A \vee c=\blacksquare$
$\vee\ \exists i.\ t_i=\blacktriangle$
s
c
t_1
...
t_n
P_0
...
tokens received from all children
repeats to all children
rules for the internal nodes:
$T2.up$
I
c
t_1
...
t_n
t = *if* $c=\blacksquare \vee \exists i.\ t_i=\blacktriangle$
then ▲ *else* △
I
...
tokens received from all children
$T2.down$
...
...
repeats to all children
rules for the leaves:
$T3.up$
(initially)
I
△
△
I
$T3.down$
△

FIGURE 2. Rules of the termination detection algorithm. The status s of a node may be either *active* (A) or *idle* (I). The color c of a node may be either *white* (□) or *black* (■). A token may be either *white* (△) or *black* (▲). ● is a repeat.

*Rule T*1*.down* is for the root of the spanning tree. When the root P_0 has received a token from each of its children, then P_0 destroys those tokens, becomes white, and sends a repeat to each of its children iff either (i) P_0 is active, or (ii) P_0 is black, or (iii) P_0 has received a black token.

*Rule T*2*.up* is for any internal node of the spanning tree (neither the root nor the leaves). When an internal node P_i has received a token from each of its children and P_i is idle, then P_i destroys those tokens and sends a new token t to its parent. The token t is black iff either P_i is black or any of the tokens received from the children is black, otherwise the token is white.

Rule $T2.down$ is for any internal node of the spanning tree (neither the root nor the leaves). When the internal node has received a repeat, it destroys that repeat and sends a repeat to each of its children.

Rule $T3.up$ is for any leaf. A leaf sends its white token to its parent iff the leaf is idle.

Rule $T3.down$ is for any leaf. A leaf which receives a repeat, destroys it, and acquires a white token.
Initially:
- all nodes are white;
- at least one node is active (and thus, it may become black);
- each leaf has a white token and every other node has neither tokens nor repeats.

These are *invariants* of the algorithm:
- an active node does *not* send any token (see Figure 2) and thus, it is impossible for the termination detection algorithm to indefinitely delay the main computation;
- when a node sends a token it is left without tokens, and analogously, when it sends a repeat it is left without repeats;
- tokens at the leaves are always white.

We will show below that termination is detected by the process P_0 at the root when at a given time $\bar{t}$:
(i) the process P_0 at the root is idle, and
(ii) the color of the root is white, and
(iii) there is a time t before $\bar{t}$ such that in the interval $[t, \bar{t}]$ every child of the root has sent to the root one token only and that token is white (it is $\triangle$).

If Conditions (i), (ii), and (iii) above hold, then the rule $T1.down$ does not fire and no new repeat wave is generated. At this point P_0 has detected termination and it may tell all other processes to halt. We do not describe here how this last communication may be realized by suitable messages sent along the arcs of the spanning tree.

The choice between the firing of the rules of the main computation and those of the termination detection algorithm is *nondeterministic.* Thus, the main computation need not be delayed by the termination detection algorithm.

The algorithm is correct independently of the fact that processes have buffers to store the incoming messages and signals, provided the delay between sending and receiving a message or a signal is sufficiently small, that is, the propagation along the arcs of the spanning tree is sufficiently fast.

The proof of correctness of the termination detection algorithm is as follows [**1**]. We first show partial correctness. We need the following definition.

Definition 0.1. Given an *undirected, connected graph* G and a spanning tree T of G, a node n of G is said to be *outside a set* A of nodes of G iff $n \notin A$ and the path from the root of T to n (including the root) contains an element of A.

In this definition it is irrelevant whether or not the node n is included in the path from the root of T to n.

Let S be the set of nodes of the spanning tree T with one or more tokens, regardless of the colors of the tokens. Let us consider the invariant *Inv* defined as follows:

$$\mathit{Inv} \equiv \quad (\text{all nodes outside } S \text{ are idle} \vee \text{some nodes not outside } S \text{ is black} \\ \vee \text{some node in } S \text{ has a black token})$$

If we take S to be the set of all leaves, we have that the invariant *Inv* is initially true because no node is outside S. The invariant *Inv* is maintained true for each firing of the rules of Figures 1 and 2 by assuming that the set S is modified by the rules $T2.\mathit{up}$ and $T3.\mathit{up}$ only, and it is modified as follows:

the node n which sends the token is removed from S, and the parent p of that node is added to S if it is not already an element of S, that is, $S := (S \cup \{p\}) - \{n\}$.

Now, since $S = \{\text{root}\}$ implies that all nodes but the root, are outside S, we have that:

$$(\mathit{Inv} \wedge S = \{\text{root}\} \wedge \text{the root is white and idle} \wedge \text{all tokens at the root are white}) \\ \Rightarrow \text{all nodes are idle (that is, the main computation has terminated)}$$

Moreover, since *Inv* is preserved during the firing of the rules of Figures 1 and 2, the main computation has terminated if $S = \{\text{root}\} \wedge$ the root is white and idle $\wedge$ all tokens at the root are white.

The termination detection algorithm terminates because:

(i) to test the firing conditions of each rule of Figures 1 and 2 takes a finite amount of time, and

(ii) if all processes are idle then a finite number of firings of the rules of Figure 2 is sufficient to allow process P_0 to detect termination.

As indicated in [**1**], we have that the complexity of the termination detection algorithm is $O(N \times m)$, where N is the number of nodes in the graph G (that is, the number of processes) and m is the number of messages generated by the main computation according to rule $M1$.

Bibliography

[1] R. W. Topor. Termination detection for distributed computations. *Information Processing Letters*, 18(1):33–36, 1984.