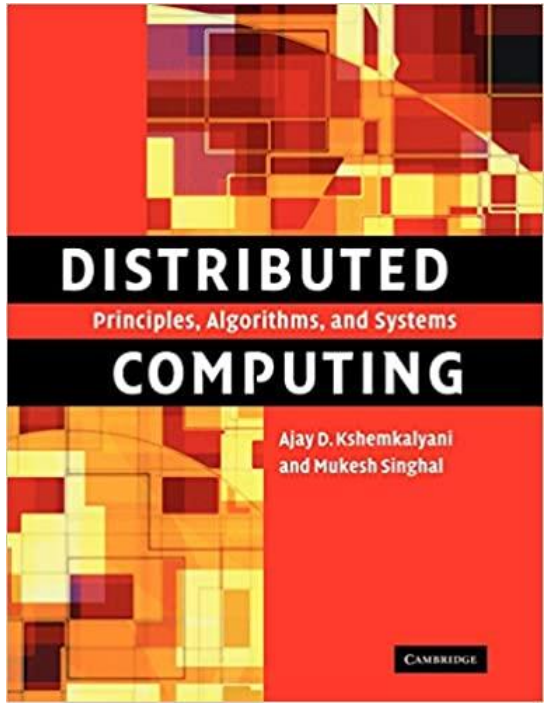
Termination Detection



System model

- At any given time, a process can be in only one of the two states: active, where it is doing local computation and idle, where the process has (temporarily) finished the execution of its local computation and will be reactivated only on the receipt of a message from another process.

- An active process can become idle at any time.

- An idle process can become active only on the receipt of a message from another process.

Termination detection problem

- To determine if a distributed computation has terminated is a non-trivial task since no process has complete knowledge of the global state, and global time does not exist.
- A distributed computation is globally terminated if every process is idle (locally terminated) and there is no message in transit between any processes.
- In the termination detection problem, a particular process (or all of the processes) must infer when the underlying computation has terminated.

Definition of termination detection

- Let $P_i(t)$ denote the state (active or idle) of process P_i at instant t.
- Let $C_{i,j}(t)$ denote the number of messages in transit in the channel at instant t from process P_i to process P_j .

- A distributed computation is said to be terminated at time instant t_0 iff:

$$(\forall i :: P_i(t_0) = \text{idle}) \text{ and } (\forall i , j :: C_{i,j}(t_0)=0)$$

- Thus, a distributed computation has terminated iff all processes have become idle and there is no message in transit in any channel.

Termination detection algorithms

- A termination detection (TD) algorithm is superposed on an underlying basic computation.

- The execution of a TD algorithm should not:
  1. affect the underlying computation (in particular, ‘freezing’ is not allowed)
  2. require addition of new communication channels between processes.

- Messages used in the underlying computation are called basic messages, and messages used for the purpose of termination detection are called control messages.

Termination Detection for Diffusing Computations

Dijkstra, E. and Scholten, C. (1980).

Information Processing Letters, Vol. 11, No. 1, pp. 1-4.

Calcul diffusant

Edsger W. Dijkstra

1972 Turing award - For fundamental contributions to programming as a high, intellectual challenge; for eloquent insistence and practical demonstration that programs should be composed correctly, not just debugged into correctness; for illuminating perception of problems at the foundations of program design.

Diffusing computation

The activity of a finite computation may propagate over a network of machines, when machines may delegate (repeatedly) subtasks to their neighbors; when such a computation is fired from a single machine, we call it a **diffusing computation**.

Superimposed signaling scheme

A signaling scheme –to be **superimposed** on the diffusing computation- enables the machine that fired it (source, initiator, environment) to detect its termination.

The signaling scheme is perfectly general in the sense that, independently of the topology of the network, it is applicable to any diffusing computation.

Superimposed signaling scheme

An existing diffusing computation can be trivially modified to pick up the capability for termination detection: for each standard message (“task to do”) used by the diffusing computation add a signal (“done”).

For each edge, define its “deficit” as the number of messages transmitted along it minus the number of signals returned along it.

Invariants (by node)

- P0: each edge has a non-negative deficit
- P1: $C \geq 0$ // sum of the deficits of its incoming edges
- P2: $D \geq 0$ // sum of the deficits of its outgoing edges

- Idle: $C = 0$ and $D = 0$

- P3: $C > 0$ or $D = 0$ // for each internal node

Cornet

The algorithm uses a cornet (pointed bag) of pending signal destinations

Very first in, very last out

Ensure that when the environment has returned to the idle state, the diffusing computation has terminated

Algorithm – state

```
INTEGER
        C               initial (0),        // sum of deficits incoming edges
        D               initial (0);        // sum of deficits outgoing edges

// Cornet
INTEGER
        engager         initial(0);
BAG OF INTEGER                              // multi set
        others          initial(∅);
```

Algorithm – procedures

1. When P_e wants to send a message to P_i // environment

   D := D + 1;

   send (message, e) to P_i

Algorithm – procedures

2. When P_j receives (message, i)

    if C = 0 then

        engager := i

    else

        others := others U i

    C := C + 1

Algorithm – procedures

3. When P_j wants to send a message to P_k

   C > 0 →

   D := D + 1;

   send (message, j) to P_k

Algorithm – procedures

4. When P_i receives (signal)

   D := D - 1

Algorithm – procedures

5. When P_j wants to send a signal

```
C > 1  or (C = 1 and D = 0) →
        if C > 1
                send (signal) to others.remove()
        else   // C=1 (and D=0)
                send (signal) to engager;  engager := 0
        C := C - 1
```

Algorithm – procedures

6. When P_e receives (signal) // environment

```
D := D - 1;
if D = 0
        // termination detected
```

Termination Detection for Distributed Computations

Topor, R.W. (1984)

Information Processing Letters, Vol. 18, No. 1, pp. 33-36.

Jeton coloré

A spanning-tree-based algorithm

- There are N processes P_i , $0 \leq i < N$, at the nodes of a fixed connected undirected graph.
- The edges of the graph represent communication channels.
- The algorithm uses a fixed **spanning tree of the graph** (no new channels are added) with process P_0 at its root which is responsible for termination detection.

A spanning-tree-based algorithm

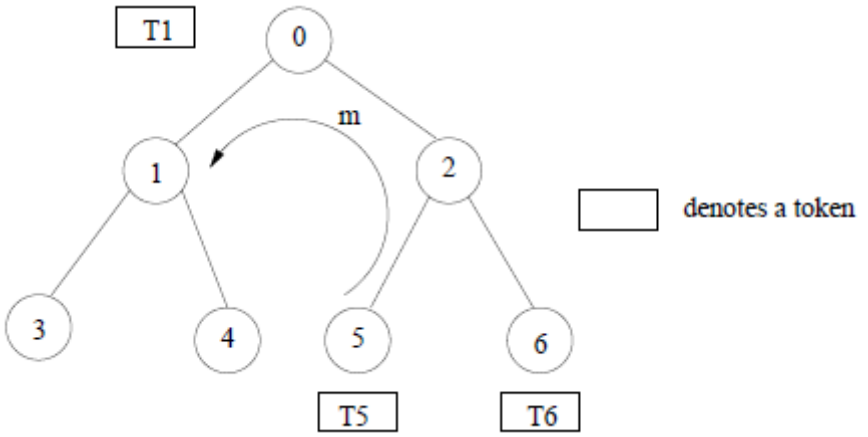
- Process P_0 communicates with other processes to determine their states (active or idle) through ***signals*** (control messages).
- All leaf nodes report to their parents, if they have terminated.
- A parent node will similarly report to its parent when it has completed processing and all of its immediate children have terminated, and so on.
- The root concludes that termination has occurred, if it has terminated and all of its immediate children have also terminated.

A spanning-tree-based algorithm

- Initially, each leaf node is given a *token*.
- Each leaf node, after it has terminated sends its *token* to its parent.
- When a parent node terminates and after it has received a *token* from each of its children, it sends a *token* to its parent.
- This way, each node indicates to its parent node that the subtree below it has become idle.
- *Tokens* are propagated to the root in a similar manner.
- The root concludes that termination has occurred if
  1. it is idle, and
  2. it received a *token* from each of its children.

A problem with the algorithm

- This simple algorithm fails when a process, after it has sent a *token* to its parent, receives a message from some other process which could cause the process to again become active.



- Hence, the root node just because it received a *token* from a child, can't conclude that all processes in the child's subtree have terminated.

Solution:

To color the processes and the tokens

Based on:

Dijkstra, E. and Scholten, C. (1980). Termination detection for diffusing computations.
Information Processing Letters, Vol. 11, No. 1, pp. 1-4.
Dijkstra, E., Feijen, W. and van Gasteren, A. (1983). Derivation of a termination detection algorithm for distributed computations.
Information Processing Letters, Vol. 16, No. 5, pp. 217-219.

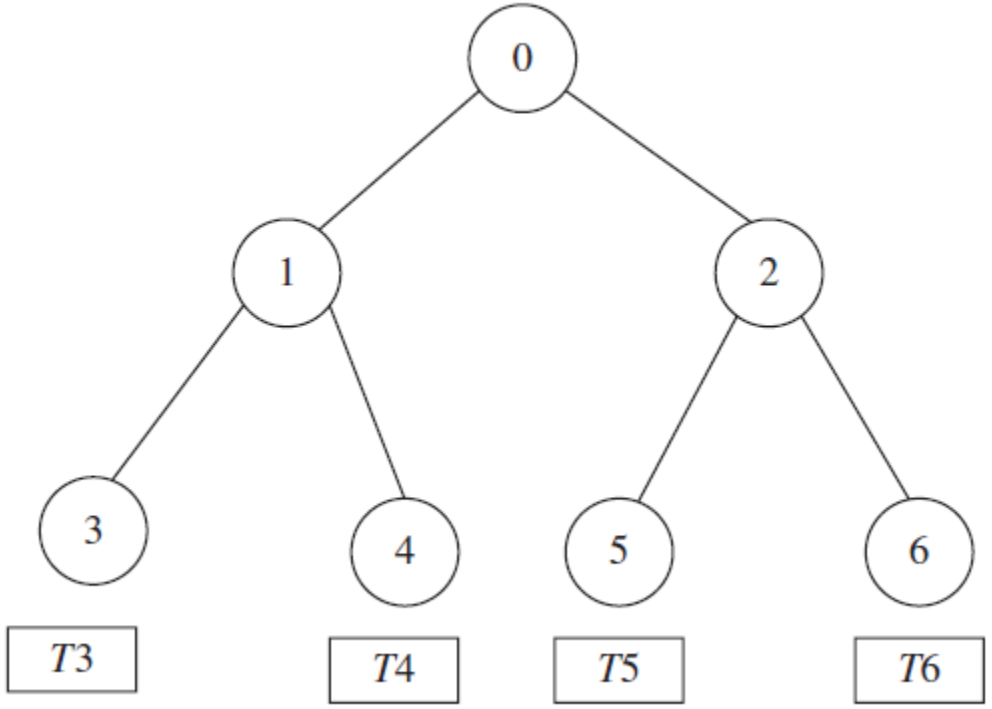
A spanning-tree-based algorithm

- Initially, each leaf node is given a *token*.
- Initially, all processes and *tokens* are colored white.
- Each leaf node, after it has terminated sends its *token* to its parent.
- When a parent node terminates and after it has received a *token* from each of its children, it sends a *token* to its parent.
- A node sending a message becomes black.
- A node that is black or has a black *token* transmits a black *token*, otherwise it transmits a white *token* (to indicate that a message-passing was involved in its subtree).
- A node transmitting a *token* becomes white.

A spanning-tree-based algorithm

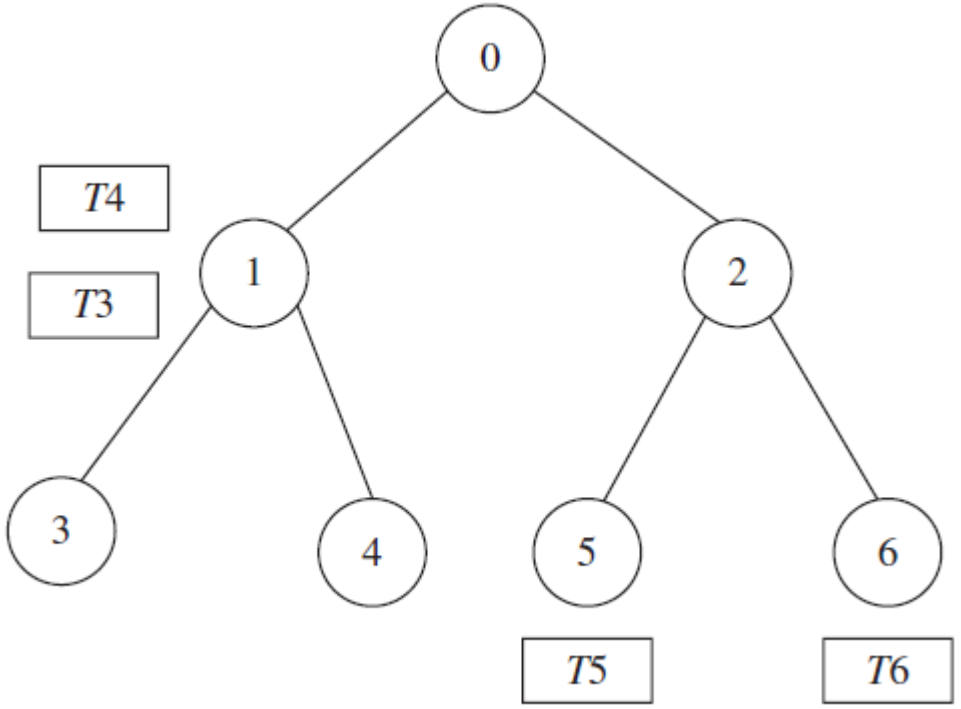
- *Tokens* are propagated to the root in this fashion.
- If node 0 has received a token from each of its children, and it is active or black or has a black *token*, it becomes white, loses its *tokens*, and sends a repeat signal to each of its children.
- An internal node receiving a *repeat* signal transmits the signal to each of its children.
- A leaf receiving a *repeat* signal is given a white *token* (and restarts the algorithm).
- The root concludes that termination has occurred, if
  1. it is idle,
  2. it is white, and
  3. it received a white *token* from each of its children.

An example



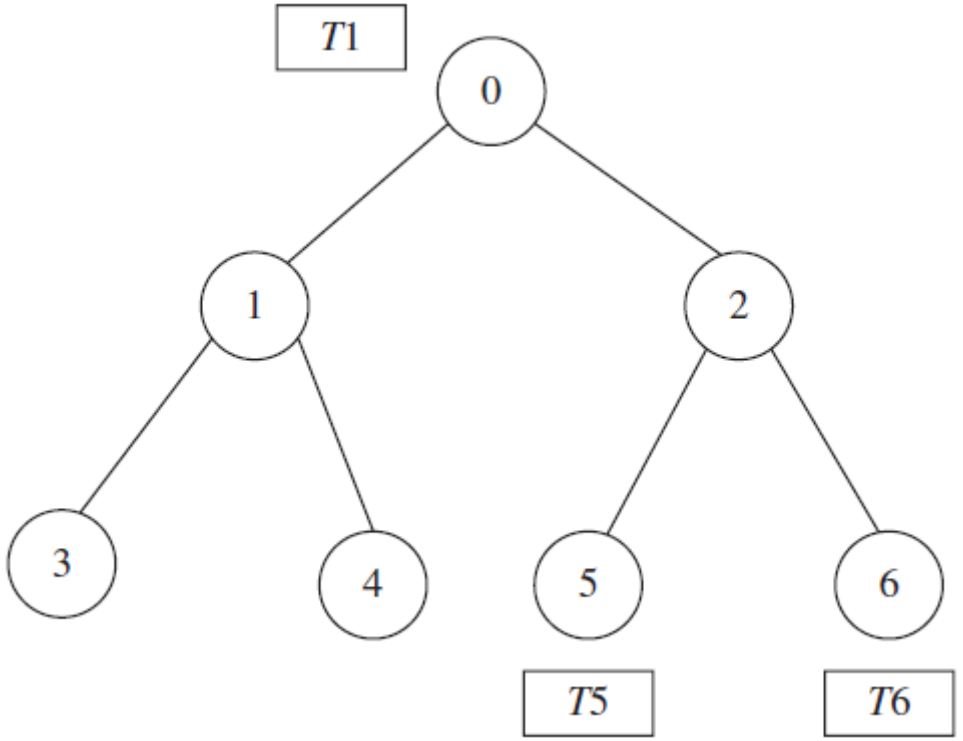
All leaf nodes have tokens. S = {3, 4, 5, 6}.

An example



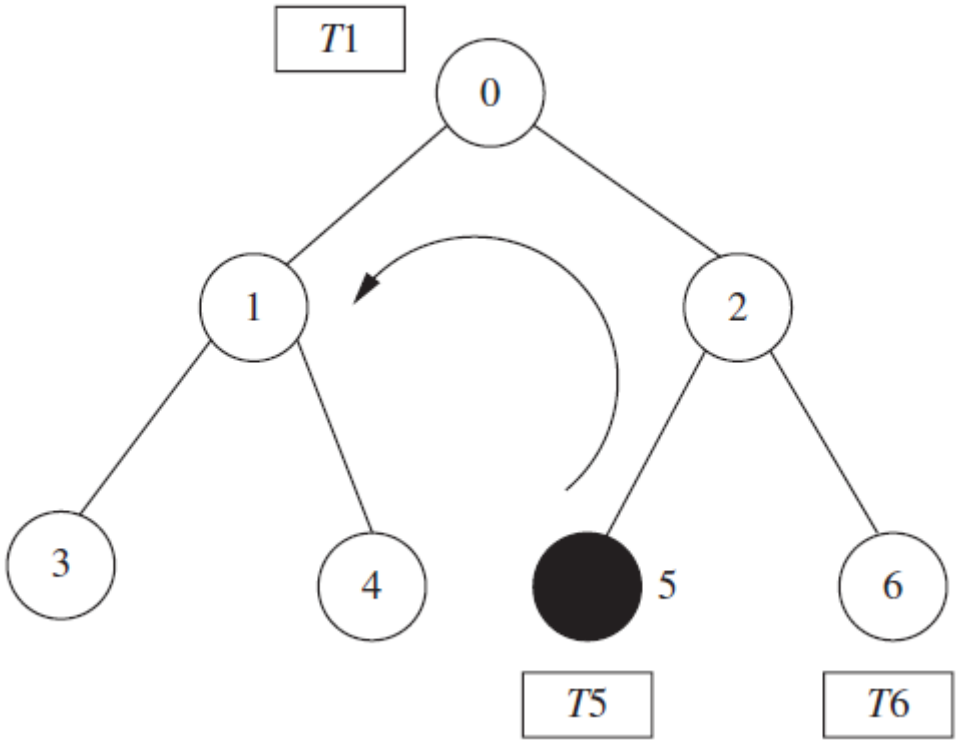
Nodes 3 and 4 become idle. S = {1, 5, 6}.

An example



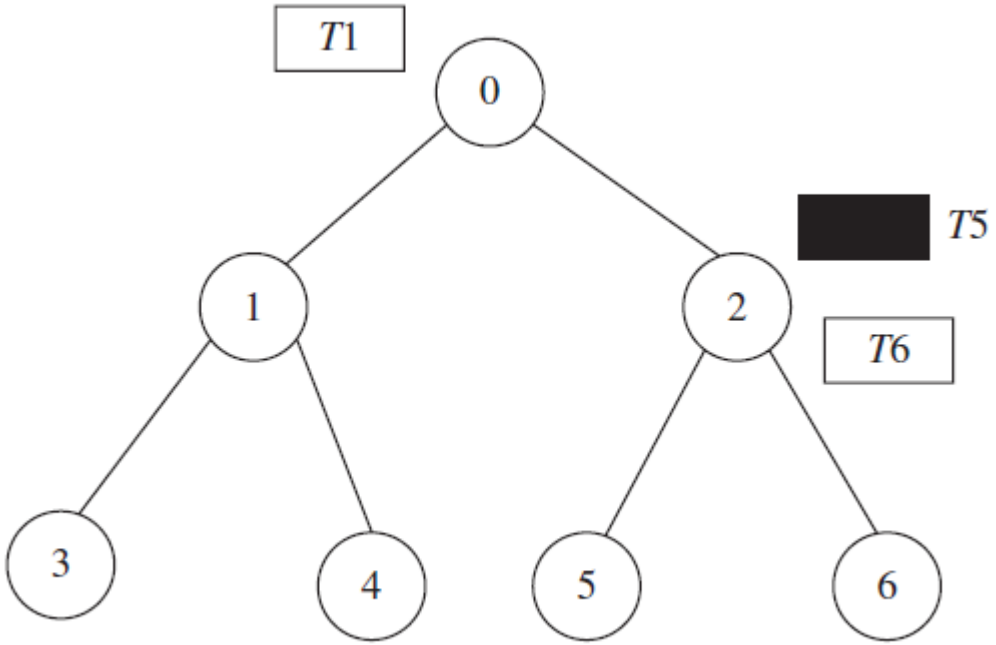
Node 1 becomes idle. S = {0, 5, 6}.

An example



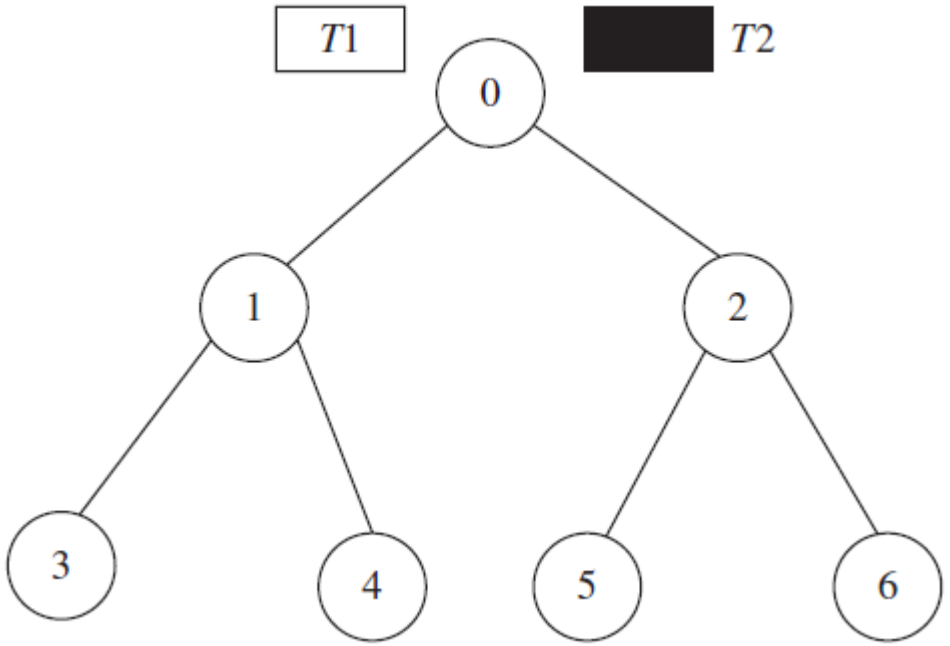
Node 5 sends a message to node 1.

An example



Nodes 5 and 6 become idle. S = {0, 2}.

An example



Node 2 becomes idle. S = 0. Node 0 initiates a repeat signal.

Invariants

- The set of nodes with one or more *tokens* at any instant is called S.
- A node j is said to be *outside* S if j is not in S and the path (in the tree) from the root to j contains an element of S.
- The algorithm is designed to maintain an invariant **R** *such that*

  R $\wedge$ S={0} $\wedge$ node 0 is idle $\Rightarrow$ all nodes are idle

- **R** is developed by a sequence of approximations.

Invariant

- **R0 ≡ all nodes outside S are idle.** **R** = R0

We can easily establish R0 by giving a *token* to each leaf and thus setting S to be the set of leaves.

Since no other node has a *token*, no nodes are outside S (the path form the root to any internal node j does not contain an element of S), and R0 is vacuously true.

Rule (of the algorithm)

The set of tokens is then moved inward according to the following rule.

O. An idle leaf that has a *token* transmits a *token* to its parent; an idle internal node that has received a *token* from each of its children transmits a *token* to its parent; an active node does not transmit a *token*. When a node transmits a *token*, it is left without any *tokens*.

A node transmitting a *token* is thus removed from S and its parent is added to S if it is not already present.

Invariant

In the absence of (basic) messages, Rule 0 alone would allow the root to detect termination.

R0 becomes false, however, if an active node not outside S sends a message to a node j outside S, and j consequently becomes active. By assuming all nodes are initially white, we can distinguish such an active node by making it black. Accordingly, let **R** = R0 ∨ R1, where

- **R1 ≡ some node not outside S is black.**

Rule

As a node sending a message has no knowledge of the position of node j with respect to S, we ensure that any message that may falsify R0 maintains the truth of R1 by the following rule.

1. A node sending a message becomes black.

Invariant

Now, R1 (hence possibly R0 ∨ R1) becomes false if the only black node is idle, is in S, and becomes outside S by transmitting a *token* to its parent.

By assuming leaves have white *tokens*, we can distinguish such a transmitted *token* by making it black. Finally, let **R** = R0 ∨ R1 ∨ R2, where

- **R2 ≡ some node in S has a black token.**

Rule

We ensure that any *token* transmission that may falsify R1 maintains the truth of R2 by the following rule.

2. A node that is black or has a black *token* transmits a black *token*, otherwise it transmits a white *token*.

Rule

As the truth of **R** does not depend on the color of nodes outside S, and as black nodes transmit black *tokens*, the truth of **R** is maintained by the following rule.

3. A node transmitting a *token* becomes white.

This rule prevents successive ***token* waves** from repeatedly transmitting black *tokens*, and is hence necessary for the algorithm to detect termination in general.

Invariant

If the token wave reaches the root, we have

R ∧ S={0} ∧ node 0 is white and idle ∧ all tokens at node 0 are white ⇒ all nodes are idle

If the hypotheses of this implication are satisfied, the main computation has terminated, and node 0 can take appropriate action such as signaling the other processes to halt.

Rule

If the hypotheses of this implication are not satisfied, node 0 initiates the outward ***repeat* wave** of the algorithm.

4. If node 0 has received a token from each of its children, and it is active or black or has a black *token*, it becomes white, loses its *tokens*, and sends a repeat signal to each of its children.

Invariant

After application of Rule 4, S is empty, so no nodes are outside S, and R0, hence **R**, is vacuously true.

Rule

To propagate the *repeat* wave outward, we apply the following rule:

5. An internal node receiving a *repeat* signal transmits the signal to each of its children.

Rule

Finally, to restore the initial states of leaves, we add the following rule:

6. A leaf receiving a *repeat* signal is given a white *token*.

As each leaf receives a *repeat* signal, it returns to its initial state (with respect to the algorithm), is added to S and restarts the inward motion of the *token* wave.

A Distributed Solution of the Distributed Termination Problem

Rana, S.P. (1983)

Information Processing Letters, Vol. 17, No. 1, pp. 43-46.

Circuit des processus et estampillage

Distributed program P = CSP

$P:: [P_1 \text{ II } \ldots \text{ II } P_n]$

$P_i :: * [S_i]$

A necessary condition for a process P_i to terminate is that it satisfies a local predicate B_i

Each process has the ability to note down the clock time whenever it satisfies its local predicate

$P_i :: B_i := false;$

$*[S_i$

$\quad \square B_i \rightarrow BTIME_i := CLOCK\text{-}TIME$

$]$

P_i waits in its top level for receipt of basic messages from other processes

Whenever P_i receives a message, B_i becomes false

Any process may initiate a detection wave over a predefined hamiltonian cycle

$P_i :: B_i := false;$

*[S_i

□ $B_i \rightarrow BTIME_i := CLOCK\text{-}TIME; P_{i+1}!detection\text{-}message\ (BTIME_i, 1)$

□ $P_{i-1}?detection\text{-}message\ (TIME, COUNT) \rightarrow$

[$\neg B_i \rightarrow$ purge the message

□ $B_i; TIME < BTIME_i \rightarrow$ purge the message

□ $B_i; TIME \geq BTIME_i \rightarrow P_{i+1}!detection\text{-}message\ (TIME, COUNT+1)$]

]

The last processes to satisfy their local predicate would succeed in detecting the termination

P_i :: B_i := false;
*[S_i
□ B_i → $BTIME_i$:= CLOCK-TIME; P_{i+1}!detection-message ($BTIME_i$, 1)
□ P_{i-1}?detection-message (TIME, COUNT) →
[COUNT = n → P_{i+1}!termination-message; TERMINATE
□ COUNT ≠ n →
[¬B_i purge the message
□ B_i →
[TIME < $BTIME_i$ → purge the message
□ TIME ≥ $BTIME_i$ → P_{i+1}!detection-message (TIME, COUNT+1)
]]]
□ P_{i-1}?termination-message → P_{i+1}!termination-message; TERMINATE
]

It is a fully distributed and symmetric algorithm

Each process follows an identical protocol and that there is no specialized process in P

Correctness

Liveness - If *the global termination condition is satisfied, termination will be eventually detected*

Suppose that P_i is the latest process to satisfy its local predicate B_i at time t.

Consider the detection message of P_i having the timestamp t. Since all processes have satisfied their local predicates for the last time at or earlier than time t, the detection message of P_i will not be purged by any process.

Thus, P_i will eventually get its detection message back and the termination will be detected.

Correctness

Safety - *There is no possibility of detecting false termination*

Detecting false termination implies getting a detection message with COUNT=n, even when the global termination condition is not satisfied.

Correctness

Consider a detection message with timestamp t forwarded by a process P_i. This implies that P_i became passive for the last time at or earlier than time t. Process P_i may again become active if some active process, say P_j initiates basic communication with it.

If P_j has already forwarded the detection message with timestamp t, then P_j was passive at time t and was made active later by another process. Continuing the chain of argument like this, we find that there is an active process P_k which communicates with a passive process who has already forwarded the detection message with timestamp t; whereas P_k itself has not received the above detection message so far.

Correctness

It is clear that P_k will purge the detection message when it reaches him because P_k was active at a time equal to or later than t.

Thus, **a detection message initiated at time t never reaches its originator if there is a single process that has been active after time t**.

Hence there is no possibility of detecting a false termination.

The last processes to satisfy their local predicate would succeed in detecting the termination

P_i :: B_i := false;

*[S_i

□ B_i → $BTIME_i$:= CLOCK-TIME; P_{i+1}!detection-message ($BTIME_i$, 1)

□ P_{i-1}?detection-message (TIME, COUNT) →

[¬B_i → purge the message

□ B_i; TIME < $BTIME_i$ → purge the message

□ B_i; TIME ≥ $BTIME_i$; COUNT ≠ n →

P_{i+1}!detection-message (TIME, COUNT+1)

□ B_i; TIME ≥ $BTIME_i$; COUNT = n →

P_{i+1}!termination-message; TERMINATE]

□ P_{i-1}?termination-message → P_{i+1}!termination-message; TERMINATE

]