Information Processing Letters 18 (1984) 33–36
North-Holland

20 January 1984

TERMINATION DETECTION FOR DISTRIBUTED COMPUTATIONS

Rodney W. TOPOR

Department of Computer Science, University of Melbourne, Parkville, Victoria 3052, Australia

Communicated by P.C. Poole
Received 17 April 1983
Revised 2 August 1983

A simple derivation of a general solution to the problem of detecting the termination of a distributed computation is presented.

Keywords: Distributed computation, distributed programming, concurrency, CSP, termination detection, program derivation

1. Introduction

An interesting and difficult problem in distributed programming is that of detecting when a distributed computation has terminated. We may describe one form of this problem as follows. Suppose we are given N *processes* P_i, $0 \leqslant i < N$, at the nodes of a fixed connected undirected graph. The processes are engaged upon a *main computation* which may require processes to send *messages* to neighbouring processes in the graph. The edges of the graph thus represent communication channels. Each process is either *active* (is still engaged on the main computation) or *idle* (has, possibly temporarily, completed its part of the main computation). The computation satisfies the following conditions:

(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.

If all processes are idle, no further computation is possible, and the main computation is said to have *terminated.*

A solution to the problem of detecting the termination of such a distributed computation is an algorithm by which a given process, P_0 say, can recognize that termination has occurred.

Such an algorithm requires process P_0 to communicate with the other processes as it cannot otherwise determine their states. Messages used for this purpose are called *signals.* The implementation of this algorithm may require each process to be modified. A desirable solution is one that satisfies the following restrictions:

(1) The modification of each process is independent of the process's definition.

(2) Execution of the algorithm cannot indefinitely delay the main computation, that is, 'freezing' is not allowed.

(3) No new communication channels between processes are added.

Solutions to this problem have previously been presented by Dijkstra [1] (without restriction (3)), Francez [2] (without restriction (2)), and Francez and Rodeh [3]. We now derive a solution satisfying all three restrictions.

2. Solution

Our solution borrows ideas from both [1] and [3]. As in [3], we use a fixed spanning tree of the given graph with process P_0 at its root and waves of signals moving inward and outward through the tree. The root of this tree plays the termination detection role of node 0 in [1], and a set of tokens moving inward from the root replaces the single token moving around the ring in [1]. The derivation of our solution follows the presentation in [1]. Henceforth we identify a node by the index of the process at that node.

The (termination detection) algorithm generates two waves of signals moving through the spanning tree. Initially, a contracting wave of signals, called *tokens*, moves inward from the leaves to the root. If this token wave reaches the root without discovering that termination has occurred, the root initiates a second, expanding, wave of *repeat* signals. As this repeat wave reaches the leaves, the token wave is gradually reformed and starts moving inward again. This sequence of events is repeated until termination is detected.

The set of nodes with one or more tokens at any instant is called S. To describe those nodes through which the token wave has passed we give the following definition.

Definition. A node j is said to be *outside* a set of nodes S if $j \notin S$ and the path (in the tree) from the root to j contains an element of S.

In general, not every path from the root to a leaf contains a node of S.

Our algorithm is designed to maintain an invariant R such that

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

We develop R by a sequence of approximations, and describe the algorithm by a set of rules which nodes may apply whenever possible.

Initially, let R be R0, where

R0 ≡ all nodes outside S are idle.

We can easily establish R0 by giving a token to each leaf and thus setting S to be the set of leaves. The set of tokens is then moved inward according to the following rule.

Rule 0. 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.

In the absence of 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 be R0 ∨ R1, where

R1 ≡ some node not outside S is black.

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.

Rule 1. A node sending a message becomes black.

Now R1, hence possibly R0 ∨ R1, becomes false if the only black 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 be R0 ∨ R1 ∨ R2, where

R2 ≡ some node in S has a black token.

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

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

As the truth of R0 ∨ R1 ∨ R2 does not depend on the colour of nodes outside S, and as black nodes transmit black tokens, the truth of R0 ∨ R1 ∨ R2 is maintained by the following rule.

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.

If the token wave reaches the root, we have

$$\begin{aligned} & R \wedge S = \{0\} \\ & \quad \wedge \text{ node 0 is white and idle} \\ & \quad \wedge \text{ all tokens at node 0 are white} \\ & \qquad \Rightarrow \text{ all nodes are idle.} \end{aligned}$$

If the hypotheses of this implication are satisfied, the main computation has terminated, and node 0 can take appropriate action such as signalling the other processes to halt. Otherwise, node 0 initiates the outward repeat wave of the algorithm. It does this by applying the following.

Rule 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.

After application of Rule 4, S is empty, so no nodes are outside S, and R0, hence R0 ∨ R1 ∨ R2, is vacuously true.

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.

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 (and is hence added to S).

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

Note that the token wave may be moving inward along some branches while the repeat wave is moving outward along other branches. The correctness of the algorithm follows, however, since each rule maintains the invariance of R0 ∨ R1 ∨ R2.

Note also that the rules do not use S which was introduced solely to aid in deriving the algorithm.

The above solution is applicable whether or not processes have buffers to store incoming messages, provided the delay between sending and receiving a message is sufficiently small.

As active nodes do not transmit tokens, it is impossible for execution of the termination detection algorithm to indefinitely delay the main computation. Hence, the modified program will eventually terminate if the original one would have. As indicated in [3], this may require $O(N*M)$ signals, where M is the number of messages in the main computation. Furthermore, as the choice between execution of the main computation and the above rules is nondeterministic, the main computation need not be delayed even temporarily by the termination detection algorithm.

An alternative derivation in which the root of the tree initiates the algorithm by sending a repeat signal to each of its children when it first becomes idle leads to a similar solution.

The above algorithm can be implemented in Hoare's CSP notation [4], for example, by simple changes to each process's definition. It suffices to modify each process by (a) setting local variables whenever the process sends a message or changes state, and (b) adding guarded commands corresponding to the various rules in the top level repetitive command of the process. These changes are independent of the process's definition, allowing the programmer to ignore termination detection when solving the original problem.

3. Summary

We have derived an algorithm to detect termination of a distributed computation. The algorithm was derived by attempting to construct and maintain an invariant that, on termination, would imply the algorithm's correctness. We believe this led to an algorithm whose correctness and properties are easier to see than those of the similar algorithm derived by more informal means in [3].

It would be interesting to find an algorithm that requires fewer signals in the worst case than this one does.

Acknowledgment

The author would like to acknowledge the inspiring example of Professor Dijkstra in his lectures on program derivation, and Netty van Gasteren's suggestions and detailed comments on an earlier version of this paper.

References

[1] E.W. Dijkstra, W.H.J. Feijen and A.J.M. van Gasteren, Derivation of a termination detection algorithm for distributed computations, Inform. Process Lett. 16 (5) (1983) 217–219.

[2] N. Francez, Distributed termination, ACM Trans. Prog. Lang. Systems 2 (1) (1980) 42–55.

[3] N. Francez and M. Rodeh, Achieving distributed termination without freezing, IEEE Trans. Software Engrg. 8 (3) (1982) 287–292.

[4] C.A.R. Hoare, Communicating sequential processes, Comm. ACM 21 (8) (1978) 666–677.