Information Processing Letters 32 (1989) 113–119
North-Holland

24 August 1989

TERMINATION DETECTION BY USING DISTRIBUTED SNAPSHOTS *

Shing-Tsaan HUANG

Institute of Computer Science, National Tsing-Hua University, HsinChu, Taiwan 30043, Rep. China

Communicated by W.M. Turski
Received 8 November 1988
Revised 7 April 1989

This paper discusses termination detection by using distributed snapshots. An algorithm is presented which is a modification of a recently proposed algorithm (Huang, 1988) The presented algorithm avoids the weak point of the original algorithm, which requires a process to wait for an acknowledgment from the receiver for every basic message sent.

Keywords: Distributed computation, distributed termination, distributed snapshot

1. Introduction

The termination detection problem for distributed computations has attracted considerable interest [2–7,9,12,13] over the past years since the works of Dijkstra and Scholten [3], and Francez [5]. In our discussion, a *distributed computation* consists of a fixed set of processes which communicate with one another via message passing. Each process may be in one of the two possible states: *active* and *idle*. An active process may go idle at any time. Only active processes may send messages to others; and an idle process can be reactivated only by receiving a message. The computation is said to be *terminated* iff all the processes are idle and there is no message in transit.

The communication environment considered is as follows. There is a logical bidirectional communication channel between each process pair. The communication channels are reliable but non-FIFO, i.e., messages sent from a sender to a receiver are correctly received by the receiver in an order not necessarily the same as the sending order. Message delay is arbitrary but finite.

Chandy and Lamport [1] proposed an elegant technique, called *distributed snapshots*, for detecting stable states in distributed computations under an environment with FIFO communication channels. This technique is further studied by Lai [9], Lai and Yang [10], and Li, Radhakrishnan and Venkatesh [11] on a non-FIFO communication environment. We shall apply their results on termination detection and present an algorithm which improves a recently proposed algorithm by Huang [7].

Besides the messages used in the computation, which are called *basic* messages, the algorithms use extra messages for the purpose of termination detection. The messages used for this purpose are called *control* messages. The presented algorithm avoids the major weak point of the original one, which requires a process to wait for a receipt acknowledgment, a control message, from the receiver for each basic message sent.

Lai [9] proposed an algorithm which uses distributed snapshots, too. The presented algorithm is different from his in that his algorithm relies on a predesignated detector to detect termination, while the presented one uses a fully distributed control.

* This work was supported in part by ATC, ERSO, Industrial Technology Institute, Rep. China under Contract SF-C-010-1.

2. Previous results on distributed snapshots

Before proceeding, let us discuss some main results on distributed snapshots. Let the processes be numbered from 1 to n, and let i mean *process* i. Also let $i \rightarrow j$ denote the channel from i to j. We use the following notations:

- $p(i, t) \equiv$ the state of i at time t;
- $s(i \rightarrow j, t) \equiv$ the set of all basic messages sent from i to j by time t;
- $r(i \rightarrow j, t) \equiv$ the set of all basic messages received by j from i by time t;
- $c(i \rightarrow j, u, v) \equiv s(i \rightarrow j, u) - r(i \rightarrow j, v)$.

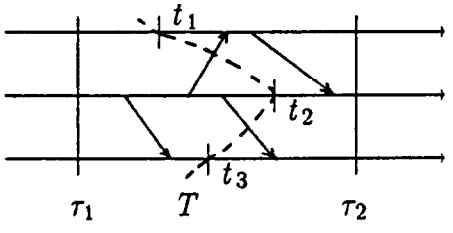
The state of a computation at time τ includes $p(i, \tau)$ for each i and $c(i \rightarrow j, \tau, \tau)$ for each $i \rightarrow j$. Note that $c(i \rightarrow j, \tau, \tau)$ is the set of all basic messages in transit on $i \rightarrow j$ at time τ, which is the channel state of $i \rightarrow j$ at time τ.

A *global snapshot* at time $T = \{t_i \mid i = 1 \ldots n\}$, denoted by $GSN(T)$, is a collection of n local snapshots with one from each i at t_i. A *local snapshot* on i at time t includes $p(i, t)$, and $s(i \rightarrow j, t)$ for each $i \rightarrow j$, $j \neq i$, and $r(k \rightarrow i, t)$ for each $k \rightarrow i$, $k \neq i$. Let $c(i \rightarrow j, t_i, t_j)$ be the state of channel $i \rightarrow j$ recorded in $GSN(T)$. Then, the state recorded in $GSN(T)$ includes $p(i, t_i)$ for each i, and $c(i \rightarrow j, t_i, t_j)$ for each $i \rightarrow j$.

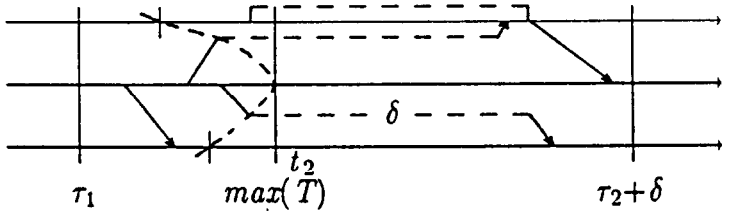
Since the local snapshots of a global snapshot are taken distributively, a global snapshot may not be feasible [10]. $GSN(T)$ is said to be *feasible* iff $(\forall i, j: : r(i \rightarrow j, t_j) \subseteq s(i \rightarrow j, t_i))$.

In this paper, we let quantified formulae have the syntactic form $(Qx, y, \ldots : R.x.y \ldots : T.x.y \ldots)$, where x, y, ... are dummies, $R.x.y \ldots$ is the range, and $T.x.y \ldots$ is the term. Since the ranges in our formulae are quite clear, we omit them.

We shall use the main theorem about distributed snapshots, which states that *all feasible global snapshots are meaningful* [1,10]. $GSN(T)$ taken between times τ_1 and τ_2 (i.e., for each i, $\tau_1 < t_i < \tau_2$) is *meaningful* iff the recorded state of $GSN(T)$ is reachable from the state of the computation at τ_1 and the state of the computation at τ_2 is reachable from the recorded state of $GSN(T)$. According to the proof in [10], this can be shown as follows. Figure 1(a) shows the original computation P. Let $\delta = \tau_2 - \tau_1$. We then construct another computation P' as shown in Fig. 1(b), which is the same as P except that every post-snapshotting event in P is now postponed for δ units of time. Note that the construction of P' is



(a) Computation P



(b) Computation P'

obtainable iff $GSN(T)$ is feasible. It is clear that in Fig. 1, the states of P and P' at time τ_1 are the same, the state of P' at time $\max\{t_1, t_2, \ldots, t_n\}$ is exactly the state recorded in $GSN(T)$ of P, and the state of P' at time $(\tau_2 + \delta)$ is the state of P at τ_2. The theorem then follows.

By the theorem, if the recorded state of $GSN(T)$ indicates a stable state, we can then conclude that the computation has reached the stable state by time τ_2 because the only possible state reachable from a stable state is the stable state itself. Since termination of the computation is a stable state, we can apply the results on termination detection.

3. The algorithm and its correctness

The main idea in the proposed algorithm is as follows. When the computation terminates, there must exist a unique process which went idle at the *latest* time. Our approach is then: when a process goes from active to idle, it issues a request to all the other processes to take a local snapshot, and requests itself to take a local snapshot, too. Upon receiving a request, if a process agrees that the requester is the latest one going idle, it grants the request by taking a local snapshot for the request. A request is said to be *successful* if all the processes have taken a local snapshot for it. We may ask the requester or any external agent to collect all the local snapshots of a request. If the request is successful, a global snapshot of the request can then be obtained. Further, if the global snapshots so obtained are feasible, then when the state recorded in a $GSN(T)$ indicates termination of the computation, viz.

$$(\forall i:: p(i, t_i) = idle) \wedge (\forall i, j:: c(i \to j, t_i, t_j) = \emptyset),$$

the termination is detected.

To realize the above idea we need a logical timing order for the requests. We let each i maintain a *logical clock* [8] represented by a variable x, which is initialized with value zero at the start of the computation. Each process increments its x by one each time it goes idle. A basic message sent by a process at its logical time x is of the form $B(Dx := x)$; and control messages that request processes to take a local snapshot issued by process i at its logical time x are of the form $R(Dx := x, Dk := i)$. (Note that we use $Dp := q$ to mean that the parameter Dp is bound with the value of q.) Each process synchronizes its x loosely with the x's on the other processes in such a way that it is maintained to have the maximum value of Dx ever received or sent in the messages. Besides x, each process also maintains another variable k such that when the process is idle, (x, k) has the maximum value of (Dx, Dk) on all messages $R(Dx, Dk)$ ever received or sent. In comparison with the logical timing order, we let

$$(Dx, Dk) > (x, k)$$
$$\text{iff } (Dx > x) \text{ or } ((Dx = x) \text{ and } (Dk > k)),$$

i.e., the comparison tie of Dx and x is broken by the process identification numbers Dk and k.

The algorithm is represented by the following four rules. We use guarded statements in the rules. Each i applies one of the rules whenever it is applicable.

(R1): When i is active, it may send a basic message to j at any time by doing
  send a $B(Dx := x)$ to j.

(R2): Upon receiving a $B(Dx)$, i does
  let $x := \max\{Dx, x\}$;
  if (i is idle) $\to$ go active.

(R3): When i goes idle, it does
  let $x := x + 1$;
  let $k := i$;
  send an $R(Dx := x, Dk := k)$ to all the other processes;
  take a local snapshot for the request by $R(Dx, Dk)$.

(R4): Upon receiving an $R(Dx, Dk)$, i does
  [$((Dx, Dk) > (x, k)) \wedge (i$ is idle) $\to$
    let $(x, k) := (Dx, Dk)$;
    take a local snapshot for the request by $R(Dx, Dk)$;
  $\square$
  $((Dx, Dk) \leqslant (x, k)) \wedge (i$ is idle) $\to$
    do nothing;
  $\square$
  (i is active) $\to$
    let $x := \max\{Dx, x\}$].

According to the above rules, not every request can become successful. However, it is rather clear that every process will eventually take a local snapshot for the latest request when the computation is terminated, that is, the latest reqest will eventually become a successful one.

Now let us prove that any global snapshot obtained from above rules is feasible. Consider the global snapshot $GSN(T)$ for a request by $R(\xi, \kappa)$, in which i takes its local snapshot at time t_i. For any basic message $B(\xi')$ through any channel $i \to j$, we can have

$$\big(B(\xi') \in r(i \to j, t_j)\big) \text{ implies } (\xi' < \xi) \tag{1}$$

and

$$(\xi' < \xi) \text{ implies } \big(B(\xi') \in s(i \to j, t_i)\big). \tag{2}$$

By (1) and (2), we have $(\forall i, j:: r(i \to j, t_j) \subseteq s(i \to j, t_i)$; hence, $GSN(T)$ is feasible.

In order to complete the proof, we need to show that (1) and (2) hold. Let x_k denote the x of k. By **(R3)** and **(R4)**, we have (while snapshotting):

$$\text{for any } k, \text{ including } \kappa,\ k \text{ is idle and } x_k = \xi \text{ at } t_k. \tag{3}$$

According to the rules, x_i is nondecreasing in time; hence, according to **(R1)** and (3), we have (2).

Now, let us look at (1). $(B(\xi') \in r(i \to j, t_j))$ implies

$$\text{at some time } t_j',\ t_j' < t_j,\ j \text{ receives } B(\xi'). \tag{4}$$

And, according to **(R2)**, we have

$$\text{right after } t_j',\ j \text{ is active and } x_j \geq \xi'. \tag{5}$$

Hence, according to (3) and (5), we have

$$\text{from } t_j' \text{ to } t_j,\ j \text{ must apply } \textbf{(R3)} \text{ at least once.} \tag{6}$$

Hence, we have $x_j \geq \xi'$ right after t_j' by (5); j letting $x_j := x_j + 1$ according to **(R3)** at least once between t_j' and t_j by (6); and, $x_j = \xi$ at t_j by (3). Therefore, we have $\xi' < \xi$. This completes our proof.

We have presented the algorithm and shown its correctness. If the state recorded in any global snapshot indicates termination of the computation, the termination is detected. For the latest request when the computation is terminated, every process will grant the request, and hence the global snapshot of this unique request will eventually become available; its recorded state must indicate the termination of the computation.

Now let us consider its implementation. As per the remarks posted by Lai [9] and Lai and Yang [10], instead of recording the set of basic messages sent and received along a channel, we can use only one message counter without distinguishing between channels to record the state of the channels. Besides x and k, each process can maintain a counter for the total number of basic messages sent minus the total number of basic messages received. At the start of the computation, we assume that all the counters are zero and all the channels are empty. Let $y(i, t)$ be the value of the counter on i at t. A local snapshot of process i at t then includes $p(i, t)$ and $y(i, t)$. A global snapshot $GSN(T)$ then includes $p(i, t_i)$ and $y(i, t_i)$ for each i. Note that since only idle processes take local snapshots in the rules, we can simplify a local snapshot further by excluding $p(\cdot)$.

We can conclude the termination of the computation if we have a $GSN(T)$ in which $(\Sigma_i y(i, t_i) = 0)$ because, by the feasibility of $GSN(T)$, $(\Sigma_i y(i, t_i) = 0)$ implies $(\forall i, j:: c(i \to j, t_i, t_j) = \emptyset)$. The implication can be shown as follows: The feasibility of $GSN(T)$ indicates

$$\big(\forall i, j:: r(i \to j, t_j) \subseteq s(i \to j, t_i)\big),$$

which implies

$$\big(\forall i, j:: |r(i \to j, t_j)| \leq |s(i \to j, t_i)|\big). \tag{7}$$

And $(\Sigma_i y(i, t_i) = 0)$ implies

$$\Big(\sum_{i \to j} |r(i \to j, t_j)| = \sum_{i \to j} |s(i \to j, t_i)|\Big). \tag{8}$$

Equations (7) and (8) imply

$$\big(\forall i, j:: |r(i \to j, t_j)| = |s(i \to j, t_i)|\big).$$

Hence, we have $(\forall i, j:: c(i \to j, t_i, t_j) = \emptyset)$.

4. Concluding remarks

We have applied the results of distributed snapshots on the termination detection problem and presented an algorithm for it. The elegancy of Chandy and Lamport's distributed snapshots can be seen in the correctness reasoning of the algorithm. The correctness reasoning of the original algorithm [7] by an invariant technique has the drawback that finding a good invariant and proving it is not an easy job, but an art. However, the proof of the feasibility of global snapshots is easier.

Since there are a bunch of algorithms based on logical times in detecting stable states in distributed computations, the results presented should have an expected influence in the design and correctness reasoning of those algorithms.

Finally, the rules are given in a rather abstract form. We do not illustrate the ways of delivering the requests and the ways of collecting the local snapshots. Like that outlined in the remarks of [7], the proposed rules can be implemented on any physical topology among the processes, provided that there exists a logical link between any process pair. In the Appendix, we give a concrete implementation of the rules on an arbitrary connected network of processes.

Appendix

In this appendix, let us have a concrete implementation of the rules on an arbitrary connected network with more than one, but a finite number of processes. The ways of delivering the requests and the ways of collecting the local snapshots in the concrete rules follow the ideas of the *diffusing* computation discussed in [3].

Consider the latest request by messages $\mathbb{R}$ from κ when the computation is terminated, and assume that it is the only request on the network. The delivering phase is a parallel traversal of the network by $\mathbb{R}$, which tries to establish a spanning tree of the network rooted at κ. Process κ sends $\mathbb{R}$ in parallel to all its neighbors. The process from which a process i receives $\mathbb{R}$ for the first time is the *father* of i in the tree. Upon receiving the first $\mathbb{R}$, a process propagates $\mathbb{R}$ in parallel to all its neighbors except its father. The edge linking a process to its father is a *tree-edge*.

The collecting phase is an echo phase which is accomplished by messages $\mathbb{C}$. Echoes arise in two situations: when $\mathbb{R}$ reaches a sink or revisits a process. A *sink* is a process with only one neighbor. When $\mathbb{R}$ reaches a sink, the sink sends back a $\mathbb{C}$ message. Note that a sink would receive $\mathbb{R}$ only once.

When $\mathbb{R}$ revisits a process i from process j, edge (i, j) is then a *non-tree edge*; in such a case, i assumes that a *virtual* $\mathbb{C}$ has been received from j. For the process j on the opposite side of the non-tree edge (i, j), it would always receive an $\mathbb{R}$ from i, which must not be the first $\mathbb{R}$ ever received by j, and j assumes the same as i does. Thus, along each non-tree edge, two $\mathbb{R}$'s and no $\mathbb{C}$ are required.

When a non-sink process has received a $\mathbb{C}$ (including virtual $\mathbb{C}$) from each of its neighbors except its father, it sends back a $\mathbb{C}$ to its father. When the requester κ has received a $\mathbb{C}$ from each of its neighbor, the collecting phase is done. It should be clear that along each tree-edge, one $\mathbb{R}$ and one $\mathbb{C}$ are required.

If any of the processes does not cooperate and ignores the messages received, the collecting phase would never be complete. In the following concrete rules, when a process takes a local snapshot for a request, it cooperates in delivering the messages corresponding to the request and takes actions accordingly. If a process decides not to take a snapshot for a request, it ignores the request message. Hence, a collecting phase for a particular request will eventually be complete iff the request can become a successful one.

In the concrete rules, we include a variable y for the counter mentioned in Section 3. Besides y, each process also maintains another variable Y. We implement taking a local snapshot by asking a process to let $Y := y$. The messages used in the collecting phase are of the form $C(Dx := \xi, Dk := \kappa, DY)$, where DY is a parameter used to sum the Y's (i.e., to collect the local snapshots). When a collecting phase is done, the requester at the root then checks if termination of the computation can be concluded.

Besides the variables mentioned above, for each non-sink idle process, we use variable z to store the number of messages $R(Dx := x,\ Dk := k)$ sent to the neighbors minus the number of messages $C(Dx := x,\ Dk := k,\ DY)$, including virtual C's, received from the neighbors. When i is idle and z is updated to zero, i sends a $C(Dx := x,\ Dk := k,\ DY)$ to its father because it has received a $C(Dx := x,\ Dk := k,\ DY)$ from each of its neighbors except its father. We also use $NS(i)$ for the set of all the neighbors of i in the network, and f for the father of i in the tree.

The initial conditions are: $x := y := z := Y := 0$; f is undefined; for an idle i, $k := i$; and all channels are empty. The readers are referred to [3] for the correctness of the *diffusing* computation.

(I1): When i is active, it may send a basic message to j at any time by doing
  send a $B(Dx := x)$ to j;
  let $y := y + 1$.

(I2): Upon receiving a $B(Dx)$, i does
  let $x := \max\{Dx,\ x\}$;
  if (i is idle) → go active;
  let $y := y - 1$.

(I3): When i goes idle, it does
  let $x := x + 1$;
  let $k := i$;
  send an $R(Dx := x,\ Dk := k)$ to each member of $NS(i)$;
  let $z := |NS(i)|$;
  let $Y := y$. /* take a local snapshot. */

(I4): Upon receiving an $R(Dx, Dk)$ from j, i does
  [$((Dx, Dk) > (x, k)) \wedge (i$ is idle) →
    let $(x, k) := (Dx, Dk)$;
    let $Y := y$;
    /* take a local snapshot. */
    let $f := j$;
    [$(|NS(i)| = 1)$ →
    /* process i is a sink. */
      send a $C(Dx, Dk, DY := Y)$ to process f;
    □
    $(|NS(i)| \neq 1)$ →
      send an $R(Dx := x, Dk := k)$ to each member of $NS(i)$ except j;
      let $z := |NS(i)| - 1$];
  □
  $((Dx, Dk) = (x, k)) \wedge (i$ is idle) →
    let $z := z - 1$; /* assume that a virtual C has been received. */
    if $(z = 0)$ → send a $C(Dx, Dk, DY := Y)$ to process f;
  □
  $((Dx, Dk) < (x, k)) \wedge (i$ is idle) →
    do nothing;
  □
  (i is active) →
    let $x := \max\{Dx, x\}$].

(I5): Upon receiving a $C(Dx, Dk, DY)$, i does
  [$((Dx, Dk) = (x, k)) \wedge (i$ is idle) →
    let $Y := Y + DY$; /* put the local snapshots together. */
    let $z := z - 1$;
    [$(z = 0) \wedge (k = i) \wedge (Y = 0)$ →
      claim *termination of the computation;*
    □
    $(z = 0) \wedge)(k = i) \wedge (Y \neq 0)$ →
      do nothing;
    □
    $(z = 0) \wedge (k \neq i)$ →
      send a $C(Dx, Dk, DY := Y)$ to process f;
    □
    $(z \neq 0)$ →
      do nothing];
  □
  $((Dx, Dk) \neq (x, k)) \vee (i$ is active) →
    do nothing].

Acknowledgment

The author wants to express his deepest gratitude to Professor W.H.J. Feijen for this detailed comments and invaluable suggestions, especially those about the notations and proofs in the paper. The author also wants to thank his graduate student S.-C. Hsu for her careful reading in preparing the paper.

References

[1] K.M. Chandy and L. Lamport, Distributed snapshots: Determining global states of distributed systems, *ACM Trans. Comput. Systems* **3** (1985) 63–75.

[2] 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** (1983) 217–219.

[3] E.W. Dijkstra and C.S. Scholten, Termination detection for distributed computations, *Inform. Process. Lett.* **11** (1980) 1–4.

[4] O. Eriksen, A termination detection protocol and its formal verification, *J. Parallel Distrib. Comput.* **5** (1988) 82–91.

[5] N. Francez, Distributed termination, *ACM Trans. Programming Lang. Systems* **2** (1980) 42–55.

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

[7] S.-T. Huang, A fully distributed termination detection scheme, *Inform. Process. Lett.* **29** (1988) 13–18.

[8] L. Lamport, Time, clocks and tbe ordering of events in distributed systems, *Comm. ACM* **21** (1978) 558–565.

[9] T.H. Lai, Termination detection for dynamically distributed systems with non-first-in-first-out communication, *J. Parallel Distrib. Comput.* **3** (1986) 577–599.

[10] T.H. Lai and T.H. Yang, On distributed snapshots, *Inform. Process. Lett.* **25** (1987) 153–158.

[11] H.F. Li, T. Radhakrishnan and K. Venkatesh, Global state detection in non-FIFO networks, in: *Proc. 7th Conf. on Distributed Computer Systems* (1987) 364–370.

[12] S.P. Rana, A distributed solution to the distributed termination problem, *Inform Process. Lett.* **17** (1983) 43–46.

[13] R.W. Topor, Termination detection for distributed computations, *Inform. Process. Lett.* **18** (1984) 33–36.