

DETECTING TERMINATION OF DISTRIBUTED COMPUTATIONS BY EXTERNAL AGENTS

Shing-Tsaan Huang

Institute of Computer Science
National Tsing-Hua University
HsinChu, Taiwan (30043), R.O.C.

ABSTRACT

This paper presents an algorithm for detecting termination of distributed computations by an auxiliary controlling agent. The algorithm assigns a weight to each active process and to each message in transit. The controlling agent has a weight, too. The algorithm maintains an invariant that the sum of all the weights related to a computation is equal to one. The controlling agent concludes the termination when its weight equals one.

A space-efficient scheme is proposed to encode the weights such that an active process can send a very large number of messages without running out of the weight.

1. Introduction.

The termination detection problem for distributed computations has attracted considerable interest [1-6,8-9,11] over the past years since the works of Dijkstra and Scholten [2], and Francez [4]. In our discussion, a *distributed computation* consists of a set of processes, which communicate with one another via message passing. Each process may be either *active* or *idle*. An active process may become idle at any time. Only active processes may send messages to others; and, an idle process can only be reactivated by receiving a message. The computation is

said to be *terminated* iff all the processes are idle and there is no message in transit.

This paper presents an algorithm for the termination detection problem under the following assumptions. A controlling agent monitors the computation. A logical communication channel exists between each process pair and between the controlling agent and each of the processes. Messages from a sender to a receiver are correctly received by the receiver in an order not necessary the same as the sending order. Message delay is arbitrary but finite. The presented algorithm can be easily extended to work on distributed systems with dynamic nature [8,10] in the sense that the processes may be created, destroyed, or migrated from one processing element to another.

The algorithm is modified from the algorithm by Rokusawa, Ichiyoshi, Chikayama, and Nakashima [10] by applying our ideas of graph search technique reported in [7]. Adopting the *weighted throw counting* scheme used in their algorithm, our algorithm assigns a weight to each active process and to each message in transit. The controlling agent maintains a weight, too. For each weight W , we let $0 < W \leq 1$. The algorithm maintains an invariant that the sum of all the weights related to the computation is equal to one. The controlling agent concludes termination if its weight equals one.

A space-efficient scheme is proposed to encode the weights such that an active process can send a large number of messages without running out of the weight, and hence avoid requiring additional weight from the controlling agent.

This work is supported in part by ATC, ERSO, Industrial Technology Institute of the Republic of China under the Contract SF-C-010-1, and the National Science Council of the Republic of China under the Contract NSC77-0408-E007-07.

We have organized the rest of the paper as follows. The algorithm and its correctness are discussed in Section 2. Then, the encoding scheme is presented in Section 3. Historic remarks of the works in [7,10] are provided in Section 4. Finally, a summary is given in Section 5.

2. The algorithm and its correctness.

Besides the messages for the computation, which we call *basic messages*, there are messages from the processes to the controlling agent for the purpose of termination detection which are called *control messages*.

The controlling agent and each of the processes maintain a variable W for its weight. Basic messages are denoted as $B(DW)$ and control messages are denoted as $C(DW)$, where DW is a parameter of the weight.

At the beginning, all the processes are idle with their weights having value zero, and the weight of the controlling agent equals one. The computation starts when the controlling agent sends a basic message to one of the processes.

The algorithm consists of the following rules. In the rules, we assume that a weight is infinitely divisible. The implementation of such weights is discussed in the next section.

- R1:** The controlling agent or an active process may send a basic message to one of the processes, say p , at any time by doing:
 Derive W_1 and W_2 such that
 $(W_1 + W_2 = W)$, $(W_1 > 0)$, and
 $(W_2 > 0)$;
 Let $W := W_1$;
 Send a $B(DW := W_2)$ to p .
- R2:** Upon receiving a $B(DW)$, a process p does:
 Let $W := W + DW$;

(If p is idle, p is reactivated.)

- R3:** An active process may become idle at any time by doing:
 Send a $C(DW := W)$ to the controlling agent;
 Let $W := 0$;
 (The process becomes idle.)
- R4:** Upon receiving a $C(DW)$, the controlling agent does:
 Let $W := W + DW$;
 If $W = 1$, the computation is terminated.

The correctness reasoning of the algorithm is as follows. Let us define the following sets first.

- A:** the set of all the W s on active processes.
B: the set of all the W s on basic messages in transit.
C: the set of all the W s on control messages in transit.

Also, let us denote the W on the controlling agent as W_c .

The following $P1$ and $P2$ are invariants.

- $P1: W_c + \sum_{W \in (A \cup B \cup C)} W = 1.$
 $P2: \forall W \in (A \cup B \cup C), W > 0.$

Hence,
 $W_c = 1$
 \Rightarrow (by $P1$)
 $\sum_{W \in (A \cup B \cup C)} W = 0$
 \Rightarrow (by $P2$)
 $(A \cup B \cup C) = \phi$
 $\Rightarrow (A \cup B) = \phi$
 \Rightarrow The computation is terminated.

That is, the algorithm never detects a false termination.

Further,
 $(A \cup B) = \phi$
 \Rightarrow (by $P1$)

$$\begin{aligned}
& W_c + \sum_{W \in C} W = 1 \\
\Rightarrow & \text{(Message delay is finite.)} \\
& \text{Eventually, } W_c = 1.
\end{aligned}$$

That is, the algorithm detects every true termination in finite time.

We have presented an algorithm for the termination detection on distributed computations. With the assumption that a weight is infinitely divisible, the algorithm can be easily extended to work on distributed systems with dynamic nature. In a dynamic system, an active process may spawn another active process or migrate from one processing element to another. We can let an active process create another active process by splitting its weight into two parts, keeping one part as its new weight, and assigning the other part as the weight of the spawned process. Migration of the processes won't cause any problem provided that only active processes are allowed to migrate. However, the maximum number of co-existing active processes is bounded in an actual implementation. We shall readdress this further after the implementation of the weights is discussed.

3. The implementation of the weights.

In this section, we present a space-efficient scheme to encode the weights. Except W_c , the weight on the controlling agent, all the other weight W is encoded in a small number of bits, yet it behaves as if it were infinitely divisible.

We may represent the W as a binary number in an unbounded array of bits with the binary point before the first bit. That is, the array (100...) has a binary value 0.100... or a decimal value 0.5. By doing this, W_1 and W_2 in rule $R1$ are always available. For example, let $W = (00100...)$, we can have $W_1 = (00010...)$ and $W_2 = (00010...)$. The computation may start when the controlling agent lets its $W_c := (100...)$ and sends a basic message $B(DW := (100...))$ to one of the

processes. The termination is detected when W_c has a *carry* from the addition in rule $R4$.

In this way, at the beginning, one bit is sufficient to encode the weight. As the computation proceeds, we may need more bits to encode the weights, and the number of bits needed might not be affordable for the processes to maintain or the messages to carry. In the following, we present a space-efficient scheme to handle this problem.

In the scheme, the controlling agent preserves an array of a sufficiently large number of bits to encode W_c . The array is evenly divided into window slots. Each other weight W on an active process or on any message in transit is represented by $(...)_i$, where $(...)$ is a window of bits and the subscript i is its corresponding slot number. For each $W = (...)_i$, it is assumed that all the bits in windows other than slot i have value 0. This is similar to a floating-point encoding scheme, in which the number of leading zeros is stored rather than the zero's themselves.

When an active process has weight $W = (0...01)_i$ and wants to send a basic message, it can slide the window to the right one slot position, or let $i := i + 1$, and then split its weight into two non-zero parts, keeping one part as its new weight and sending the other as the weight of the message. For example, a process can split $(0...01)_0$ into $(1...11)_1$ and $(0...01)_1$, then keep $(1...11)_1$ as its new weight and assign $(0...01)_1$ as the weight of the message.

An active process may have two weights according to rule $R2$, one from itself and the other from a received basic message. If the sum of the two weights can not be fitted into a single window slot, we let the process keep the window with smaller slot number and send the other to the controlling agent. For example, when an active process having weight $= (...)_3$ receives another weight $= (...)_1$, we let it keep the weight $(...)_1$ and send $(...)_3$ to the controlling agent.

The above scheme makes possible for each process to maintain only a few bits, and each message to carry the same number of bits. For example, let H be the array size of W_c and h be the window size. If $h = 16$, and $H = 1024h$, we need 26 bits to represent the W , among them 10 bits are for the slot number because $2^{10} = 1024$, and 16 bits are for the window itself. Only the weight W_c of the controlling agent then needs 1024 16-bit computer words.

When an active process has its slot number reaching the upper limit, it can split its weight into two parts, keep one part as its new weight and send the other as a *request* for *supply* of weight to the controlling agent. The weight on a request message can be of the form $(0...01)_i$. Upon receiving such a request with weight $(0...01)_i$, the controlling agent then sends a supply of weight $(0...01)_j$ with $j < i$, to the requester, or postpones the sending of the supply until it is able to do so. When an active process receives a supply, the situation is similar to what described in the paragraph before the last paragraph, and similar actions can be followed by the process. When an idle process receives a supply, it simply rebounds the weight to the controlling agent. By enlarging H , the requests for supply of weight can be eliminated.

In a dynamic system, the fact that W_c is bounded by H also has its effects on the maximum number of active processes that can co-exist at any time. With $H = 2$, for example, the maximum number of co-existing active processes is 4 because in such a case, the minimum possible weight for each active process is binary value 0.01 and the sum of their weights is equal to one.

We have presented a space-efficient scheme to encode the weights. This scheme maintains a very large resource pool in the controlling agent which is shared by all the processes with each having only a small amount of memory space. This idea should be interesting to the community of researchers studying resource sharing problems.

4. Historic Remarks.

The presented algorithm applies the ideas of the graph searching technique reported in [7] on the weighted throw counting scheme discussed in [10]. In searching a graph with logical edges, the searching can start from a *root* with a weight of one. The root sends a token $T(DW := 1/|NS|)$ to each of its neighbors. We use NS to denote the set of the neighbors of a node. Upon receiving a token $T(DW)$ for the first time, a node sends a token $T(DW := DW/(|NS|-1))$ to each of its neighbors except the one from which the token is received. For a revisited token $T(DW)$, a node sends a $T(DW)$ to the root. A *sink* node, a node with only one neighbor, sends a $T(DW)$ to the root upon receiving a token $T(DW)$. The root detects the termination of the searching when the sum of the weights on all the returned tokens equals one. This search technique assumes that a weight is infinitely divisible.

Although different formulation and terminology are used, the termination detection dealt with in the algorithm reported in [10] is similar to the one discussed in this paper. However, in [10], the weights are integer numbers; since no special encoding scheme is used, each activated process almost always needs to request for supply of weight. As discussed in [10], let us have the following weight assignment strategy: Assign a fixed weight 2^{10} to a basic message if the weight of the sender is more than twice of that; otherwise assign half of its weight. Then, an active process which is reactivated by receiving a basic message with a weight of 2^{10} can only send 10 basic messages. In our proposed scheme, however, by sliding the window to the right, the effect is receiving a supply of a very large weight from the controlling agent.

For instance, let $H = 1024h$, and $h = 16$. Then, by sliding the window one slot position, a process can send an additional 2^{16} basic messages. The computation can go into a depth of 1024 levels without any request messages, and hence, the processes are free

from experiencing any delays waiting for supply messages. The memory cost for each process and message is 26 bits only.

The major disadvantage of the scheme is that in rule *R2* the sum of the two weights may not be able to be fitted into a single window of bits, hence an extra control message is needed. However, the advantage of the freedom of the processes from waiting for supply messages should be regarded as more important. Further, in some applications, sending basic messages to an active process can be avoided. For example, in the discussed graph searching computation, a process (node) goes active upon receiving the token, and goes idle immediately after sending out the tokens to its neighbors; therefore, no token is sent to an active process.

5. Summary.

We have presented a termination detection algorithm for distributed computations and proposed a space-efficient encoding scheme to implement the ideas that logically a weight can be infinitely divisible. The algorithm is modified from the one proposed by Rokusawa, et al. [10], and borrows the ideas from our other article [7]. Using our encoding scheme, each process and message need a small number of bits to encode the weight, and the processes can almost be free from the delays waiting for supply of weights from the controlling agent.

Since a termination detection algorithm should do the best to avoid bothering the computation, having an active process waiting for control messages is not desirable. And, the detecting delay, the period from the computation terminates until the controlling agent detects the termination, should be kept as less as possible. Further, the number of control messages sent should be as small as possible. The presented algorithm is almost optimal in these aspects. With a large enough *H*, the processes won't receive any control messages. It takes only one message delay, from the latest idled process to the controlling agent, to detect the termination. And, for every basic message, at most one control message is required for the

case that the sum of the weights in rule *R2* can not be fitted into a single window of bits. Finally, only one control message is needed for each idleness of the processes.

ACKNOWLEDGMENT

The author would like to thank the anonymous referees for their helpful comments and suggestions.

References

- [1] Dijkstra, E. W., Feijen, W. H. J., and van Gasteren, A. J. M. Derivation of a Termination Detection Algorithm for Distributed Computations. *Inform. Processing Lett.*, Vol. 16, pp. 217–219, June 1983.
- [2] Dijkstra, E. W. and Scholten, C. S. Termination Detection for Distributed Computations. *Inform. Processing Lett.*, Vol. 11, pp. 1–4, Aug. 1980.
- [3] Eriksen, O. A Termination Detection Protocol and Its Formal Verification. *J. Parallel and Distributed Computing* 5, pp. 82–91, 1988.
- [4] Francez, N. Distributed Termination. *ACM Trans. Program. Lang. Syst.*, Vol. 2, pp. 42–55, Jan. 1980.
- [5] Francez, N. and Rodeh, M. Achieving Distributed Termination without Freezing. *IEEE Trans. Software Engrg.*, Vol. 8, pp. 287–292, Mar. 1982.
- [6] Huang, S-T. A Fully Distributed Termination Detection Scheme. *Inform. Processing Lett.* Vol. 29, pp. 13–18, Sept. 1988.
- [7] ——. A distributed deadlock detection algorithm for CSP style programs. *TR-H7601*, Inst. of Computer Science, Tsing-Hua University, Feb. 1987, submitted for publication, under revision.
- [8] Lai, T-H. Termination Detection for Dynamically Distributed Systems with Non-first-in-first-out Communication. *J. Parallel and Distributed Computing* 3, pp. 577–599, 1986.
- [9] Rana, S. P. A Distributed Solution to the Distributed Termination Problem.

- Inform. Processing Lett.*, Vol. 17, pp. 43–46, July 1983.
- [10] Rokusawa, K., Ichiyoshi, N., Chikayama, T. and Nakashima, H. An efficient termination detection and abortion algorithm for distributed processing systems. *Proc. 1988 Int'l Conf. Parallel Processing*, pp.18–22, 1988.
- [11] Topor, R. W. Termination Detection for Distributed Computations. *Inform. Processing Lett.*, Vol. 18, pp. 33–36, Jan. 1984.