ICOT Technical Report: TR-0823

TR-0823

# A Scalable Termination Detection Scheme by Weighted Throw Counting with Delayed Weight Returning (Extended Abstract)

by
N. Ichiyoshi (MRI) & K. Rokusawa (Oki)

December, 1992

© 1992, ICOT

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

# A Scalable Termination Detection Scheme by Weighted Throw Counting with Delayed Weight Returning (Extended Abstract)

## 1 Introduction

Termination detection is an basic issue in distributed computation both theoretically and in practice. Unlike sequential computation, termination detection may not be trivial for parallel computation because of the difficulty in obtaining a consistent global state, especially when there can be interprocessor messages in transit.

A number of distributed termination detection algorithms have been proposed. The Weighted Throw Counting (WTC) scheme [3] is a simple and efficient algorithm based on the weighted reference counting technique for incremental distributed garbage collection [1, 4]. The idea is assigning a known amount of "weight" to the whole computation and letting it to be divided among all distributed components of computation (processes and messages), and detecting termination when all of the original weight has been returned.

In the original WTC scheme, as the number of processes grows, the detecting process can be overwhelmed by the incoming weight-return messages. Hence the original scheme is not a scalable one. This paper presents a method for removing this bottleneck, thus making the termination detection scheme more scalable, i.e., effective even if the number of processes grows very large. In contrast to message complexity commonly used as an efficiency criterion of termination detection algorithms, the scheme is aimed at minimizing the time complexity of detection.

## 2 Computational Model

The activity of the base computation on a processing node will be referred to by a *process*. We assume the following about the base computation.

1. A process can be in either of the two states: *active* or *idle*.
2. An active process $p$ can send messages to any process $q$.
3. An active process can spontaneously become idle at any time.
4. An idle process can become active only when it receives a message. (For this property, a message in the base computation is sometimes called an *activation message*.)
5. Initially, all processes are idle and there are no messages; at the start of computation, one process is activated by the outside "environment".

A distributed computation is terminated when all the processes are idle and there are no messages in transit (sent but not yet received). Termination is a stable state [2], that is, a terminated computation remains terminated for ever.

# 3 Weighted Throw Counting (WTC)

In the Weighted Throw Counting technique [3], the initially activated process is assigned a certain positive weight $W$ and the environment is given the same weight, and the following invariant condition is maintained:

- Active processes and messages have positive weights.
- Idle processes have a zero weight.
- The sum of all weights of processes and messages are equal to the weight that the environment has.

Under the above condition, the condition that the environment has weight zero is equivalent to there being no active processes and no messages in transit, that is, the base computation being terminated.

To maintain the invariant, the handling of activation messages are changed as follows. When a process sends an activation message, it splits the weight $W$ in two positive values $W_1$ and $W_2$ such that $W = W_1 + W_2$, assigns $W_1$ to the activation message and retains $W_2$ to itself. When a process receives an activation message, the weight carried by the message is added to its weight.

To actually detect termination, a detecting process is placed on some processing node. When computation is initiated by the environment, the first process and the detecting process are given the same weight. On becoming idle, a process in the base computation sends a %return_WTC ($W$) message [1] to the detecting process to return the weight it has. When the detecting process receives a %return_WTC ($W$) message, it subtracts $W$ from its weight (Fig 1).

The detecting process detects termination when its weight becomes zero. The invariant condition guarantees correctness of termination detection, and the eventual delivery of %return_WTC messages to the detecting process guarantees the liveness property.

**Example 1** *Suppose that it takes 100 μsec to service an incoming %return_WTC message. The processes alternatively becomes active and idle during computation. Suppose that, in some distributed computation, the average cycle is 10 msec, which means a process turns idle from active every 10 msec on the average. It follows that if there are more than 100 processes then the detecting process cannot keep up with the incoming %return_WTC messages, thus becoming the bottleneck.*

[1] In this paper, message names are prefixed by a percent sign.

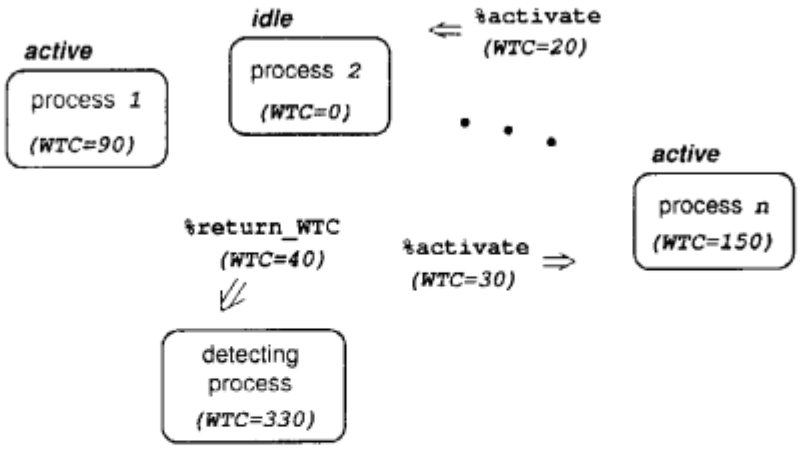


Figure 1: Weighted Throw Counting

# 4 Bottleneck in the Original WTC and Its Removal by Delayed Returning

The original WTC scheme has a potential bottleneck: As the number of processes increases, the detecting process may be flooded with %return_WTC messages simultaneously sent from a large number of processes and may become bottleneck.

We propose a *delayed (weight) returning* rule to ensure that the detecting process will not become bottleneck. Under this rule, processes are required to keep the message sending rate below a certain level so that the detecting process is not overwhelmed. Specifically, if

(1) there are $n$ processes $(p_1, \ldots, p_n)$ to a parent process,

(2) the message service rate (inverse of message service time) at the detecting process is $\lambda_r$, and

(3) the message sending rate of process $i$ is $\lambda_{s(i)}$,

then it must be that

$$\sum_{1 \le i \le n} \lambda_{s(i)} < \lambda_r,$$

or, simply,

$$\lambda_{s(i)} < \lambda_{smax} = \lambda_r / n \quad \text{for } 1 \le i \le n.$$

The rate $\lambda_{smax}$ is the maximum allowed sending rate. Its inverse, $t_{smin}$, is the minimum allowed sending interval.

A *delayed returning mechanism* implements the delayed returning rule. Here are two candidates, assuming that the message interval of $t_{smin}$ is equivalent to the message sending rate of $\lambda_{smax}$.

(1) Each non-root process keeps a local clock which schedules the message sending routine at an interval of $t_{smin}$. The routine sends a %return_WTC message if the process is idle when it is invoked.

(2) On becoming idle, each non-root process waits $t_{smin}$ before sending a %return_WTC message. If it receives an activation message before $t_{smin}$ elapses, it simply becomes active.

Two possible ways of how a process waits $t_{smin}$ is:

(a) To set an local alarm clock to wake it up after $t_{smin}$ unit of time, and to sleep. On receipt of an activation message during the sleep, the alarm is canceled.

(b) To go into a busy waiting loop when it becomes idle. The process counts down in the loop till the counter value reaches zero. The initial value of the counter is determined so that the time $t_{smin}$ elapses when the countdown finishes. On receipt of an activation message during the sleep, the busy loop is exited.

# 5 Message Combining

The delayed returning mechanism has removed the bottleneck in the original WTC scheme. However, as easily seen, when all the processes terminate at once, it takes time proportional to the number of processes to detect termination. As the number of processes grows, the detecting time might become unnegligible relative to the base computation time. This can be solved by introducing a logical combining tree. The delayed returning rule applies between each internal process and its child processes.

# 6 Time Complexity

It can be guaranteed that, by properly configuring a combining tree as the number of processes $p$ grows, the worst-case time for termination detection is of the same asymptotic order as the broadcast time for any underlying network topology.

# References


[1] D. I. Bevan. Distributed garbage collection using reference counting. In *Proceedings of Parallel Architectures and Languages Europe*, pages 176 187, June 1987. Also in *Parallel Computing*, Vol.9, No.2, pp.179–192, 1989.

[2] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[3] K. Rokusawa, N. Ichiyoshi, T. Chikayama, and H. Nakashima. An efficient termination detection and abortion algorithm for distributed processing systems. In *Proceedings of the 1988 International Conference on Parallel Processing, Vol. I Architecture*, pages 18–22, 1988.

[4] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In *Proceedings of Parallel Architectures and Languages Europe*, pages 432–443, June 1987.