ICOT Technical Report: TR-341

TR-341

An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems

by
K. Rokusawa, N. Ichiyoshi
T. Chikayama & H. Nakashima

February, 1988

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems

Kazuaki Rokusawa Nobuyuki Ichiyoshi Takashi Chikayama
Institute for New Generation Computer Technology *

Hiroshi Nakashima
Mitsubishi Electric Corporation †

Abstract

This paper describes an algorithm for termination detection and abortion in distributed processing systems, where processes may exist not only in processing elements but also in transit. The algorithm works correctly whether the communication channels are first-in-first-out or not, and no acknowledgement message is required. Assigning weights to all processes and maintaining the invariant that the sum of the weights is zero are the main features of the algorithm.

1 Introduction

Termination detection and abortion of all processes in a system are major functions in parallel processing. They are easy in closely-coupled systems, such as shared memory multiprocessors, but difficult in distributed systems, particularly when there are processes in transit.

We have devised an algorithm for termination detection and abortion in distributed processing systems, where processes may exist not only in processing elements but also in transit. This algorithm is called the *weighted throw counting* scheme, which is an application of the weighted reference counting scheme [1] [5], a garbage collection scheme for parallel processing systems.

The algorithm will be applied to parallel implementation of KL1, a parallel logic programming language based on GHC [4], on the Multi-PSI [3], a collection of Personal Sequential Inference Machines [6] (PSI's) interconnected by a fast communication network.

This paper is organized as follows. Section 2 defines the computation model employed. Section 3 shows the problems of termination detection and abortion in distributed systems. A naive solution is presented in section 4. Section 5 describes the algorithm for termination detection and abortion where the communication channels are first-in-first-out. The algorithm for the system with non-first-in-first-out communication is presented in section 6. Finally the comparison of the algorithm with the naive one is given in section 7.

2 Computation Model

The following process model is assumed:

- A process pool consists of one controlling process and a finite number of child processes;
- There are a finite number of process pools in the system;
- Each process pool is assigned a unique process pool identifier (PID);
- A child process can terminate at any time;

*Fourth Research Laboratory, Institute for New Generation Computer Technology, 4-28, Mita 1-chome, Minato-ku, Tokyo 108 JAPAN

†Computer Systems Development Department, Information Systems and Electronics Development Laboratory, Mitsubishi Electric Corporation, 1-1, Ohfuna 5-chome, Kamakura-shi, Kanagawa 247 JAPAN

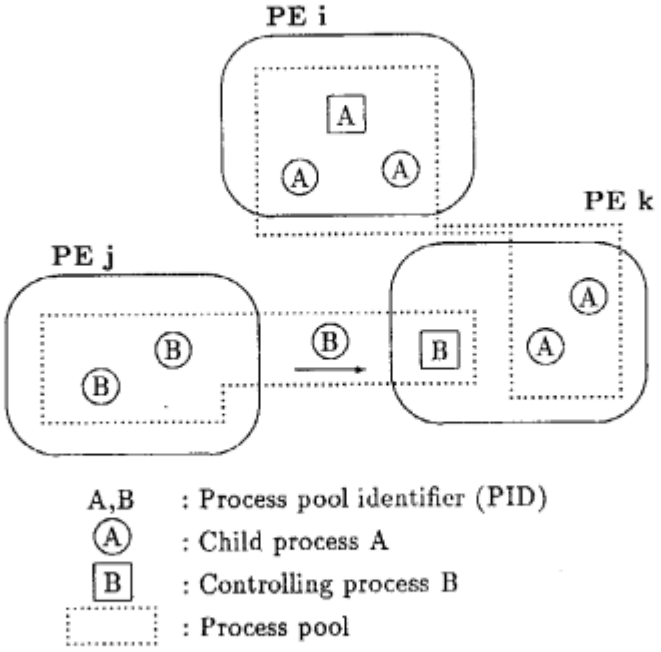

Figure 1: Computation Model

- A child process can generate another child process having the same PID and a new process pool having a new PID as well.

In this paper, "process" means "child process" unless otherwise indicated. A process pool *terminates* if all the children terminate. *Aborting* a process pool is forcing all the children to terminate. A process pool described above is distributed over the following machine:

- A finite number of processing elements (PEs) interconnected by a communication network;
- No global storage; PEs may communicate by passing messages;
- Asynchronous communication, in which messages are delivered with arbitrary finite delay.

It is assumed that a PE can detect the termination of all processes in it having the same PID and can force them to terminate. The controlling process and PEs can communicate in both directions. A PE may send a message to the controlling process informing it of the termination of all processes, and the controlling process may send a message to abort processes.

Although there exist a finite number of process pools in the system at a given time, there is no limitation of total number of process pools, since any process can generate a new process pool at any time.

Processes may migrate among PEs for load balancing. To achieve this, a PE may throw a process in the PE to another PE and the thrown process is delivered with arbitrary finite delay. Therefore, at a given time, processes may be in transit in the communication network but not in any PEs.

3 Problems

This section describes why termination detection and abortion of processes distributed over several processors are difficult, particularly when there are processes in transit.

3.1 Termination Detection

The controlling process must detect the termination of all processes having the same PID as

the controlling process.

Each PE can detect the termination of all processes with the same PID in the PE locally and can send a message indicating termination (*terminated* message) to the corresponding controlling process.

However, even if the controlling process receives *terminated* messages from all PEs, it is not sure that all processes have terminated. There may be processes in transit, which will be received by a PE after the PE has sent a *terminated* message.

3.2 Abortion

The controlling process must force to terminate all processes having the same PID as the controlling process.

If the controlling process broadcasts a message causing processes in the pool to terminate (*abort* message), it is possible to abort all the processes in the PE, but impossible to abort the processes in transit. After receiving an *abort* message and aborting the processes, the PE may receive a thrown process.

If a PE memorizes the PID carried by the *abort* message and ignores received processes with the same PID as that memorized, the *abortion by broadcast* scheme described above may work. However, this scheme has disadvantages. First, if only a few PEs have the process to be aborted, most of *abort* messages are useless. Second, it is impossible to reuse a PID, because the controlling process cannot detect the termination of the abortion; this is a major disadvantage.

4 The Naive Scheme

Ichiyoshi et al. [2] describe a termination detection scheme using *acknowledge* messages. It effectively does the following, although different terminology is used. A non-empty set of processes in one PE having the same PID forms a subpool of processes, which is called a "process subpool", or a "subpool" in short. Processes in a PE are under the control of a subpool. On receiving a thrown process, the PE decides whether there is already a subpool having the same PID as the thrown process. If there is, the PE adds the process received to the subpool and sends back an *acknowledge* message; otherwise, creates a new subpool and memorizes the sender PE of the process in it. Each subpool has a counter which is incremented on throwing a process, and is decremented on receiving the *acknowledge* message or *terminated* message. When all processes in it are terminated and the value of the counter reaches zero, the subpool terminates and sends a *terminated* message to the PE memorized.

This scheme is simple and termination can be detected correctly; if the value of the counter reaches *zero*, there is neither process thrown from the corresponding subpool in transit nor subpool created by the thrown process from the corresponding subpool. However, it has a serious disadvantage; termination of a subpool depends on terminations of other subpools. Since subpools form a tree structure, the root cannot terminate unless all its leaves terminate. In the worst case, a chain of subpools is created, where each subpool terminates sequentially.

5 The WTC Scheme

We have devised a new scheme which requires no acknowledge message and makes it possible to reuse the PID. This new scheme is the *weighted throw counting* (WTC) scheme which is an application of the weighted reference counting scheme [1] [5], a garbage collection scheme for parallel processing systems.

5.1 Termination Detection

We associate *weight* with the controlling process, each process and each subpool. The weight of a process in transit and that of a subpool are positive integers, while the weight of the controlling process is a negative integer. The WTC scheme maintains the invariant that:

> The sum of the weights is *zero*.

This ensures that the weight of the controlling process reaches *zero* if and only if all processes terminate; there is no processes neither in a PE nor in transit (see figure 2).

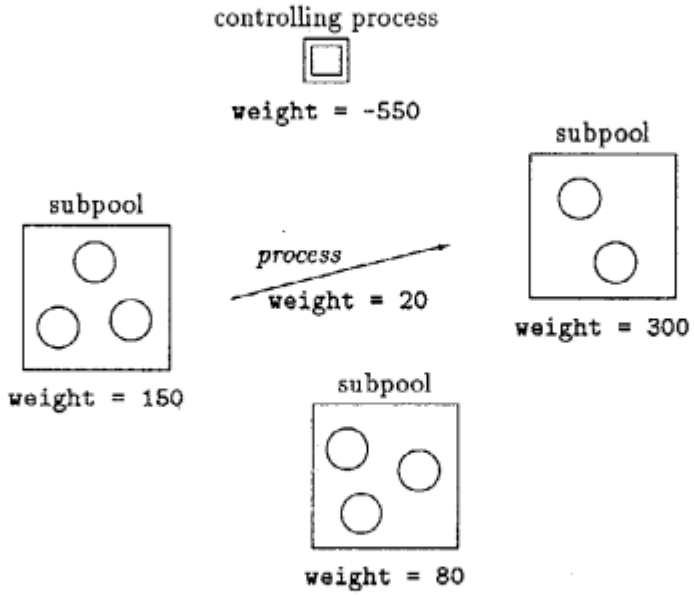

Figure 2: The WTC Scheme

When a PE throws a process from a subpool, the PE assigns a weight to the thrown process and subtracts the same amount from the weight of the subpool. The new weight of the subpool and that assigned to the thrown process should both be positive, and the sum of the two weights is equal to the original weight of the subpool. For example, if a subpool originally weighs 1000, the weight of a thrown process and the new weight of the subpool can be set to 50 and 950. When a PE receives a thrown process, it adds the weight assigned to the received process to the weight of the subpool having the same PID. If there is no subpool with the same PID, a PE creates a new subpool containing the received process and sets its initial weight at the weight of the received process.

When the weight of a subpool becomes *one*, the PE cannot throw a process, because non-zero weight must be assigned to the thrown process and non-zero weight must remain also in the subpool after throwing. The operation when this situation occurs is described in section 5.3.

When all processes in it are terminated, the subpool *terminates* and sends a *terminated* message to the corresponding controlling process. This *terminated* message gives notification of the termination of the subpool and carries the weight of the terminated subpool. On receiving a *terminated* message, the controlling process adds the weight carried by the *terminated* message to its (negative) weight. If the weight of the controlling process reaches *zero*, the termination of all processes is detected.

5.2 Abortion

This section describes an abortion scheme for the computation model with first-in-first-out communication; messages are delivered in the order sent. A scheme without this assumption is described in section 6.

The controlling process should be able to force all processes with the same PID as the controlling process to terminate, and detect the termination of all processes to reuse the PID. Termination is detected using the WTC scheme described above. Thus, only delivery of the *abort* message to each PE containing the subpool is required. To achieve this, the controlling process needs to detect creation of a subpool and to send an *abort* message to the PE containing the subpool.

We introduce here a new message, named

the *ready* message which gives notification of the creation of a subpool. On creation of a subpool, a PE sends a *ready* message to the corresponding controlling process. On receiving a *ready* message, the controlling process memorizes the sender PE, which is deleted on receiving a *terminated* message.

The controlling process performs the following operations to achieve the abortion:

(1) Sending an *abort* message to each PE memorized;

(2) Sending an *abort* message to the sender PE of a *ready* message received after operation (1).

Once the controlling process receives a *ready* message, a subpool may exist in the sender PE until a *terminated* message is received from the same PE. The controlling process therefore performs operation (1), which aborts all subpools already detected by the controlling process. Operation (2) aborts such subpools that were not recognized by the controlling process when operation (1) was carried out: a subpool that is created after operation (1), or created before operation (1) but whose *ready* message is still in transit.

It is necessary to assign a weight to an *abort* message like the thrown process, while not necessary to a *ready* message, because once the controlling process receives a *ready* message, it will receive a *terminated* message later from the sender PE of the *ready* message (the FIFO assumption).

On receiving an *abort* message, a PE performs either of the following operations:

(3a) Forcing the subpool with the specified PID to terminate, and sending back a *terminated* message which carries the sum of the weight of the terminated subpool and the *abort* message;

(3b) If there is no subpool having the specified PID, sending back a *return* message which carries back the weight assigned to the *abort* message.

Figure 3 shows the abortion operations described above.

When a subpool terminates before receiving an *abort* message, an *abort* message may reach a PE having no subpool with the same PID as the *abort* message. In this case, operation (3b) is performed and the *return* message is sent as the response to the *abort* message. On receiving a *return* message, the controlling process adds the weight of the message to its own weight. If the weight of the controlling process reaches zero by this operation, the termination of all processes is guaranteed.

During the operations of abortion, the following cyclic situation may occur. The controlling process sends an *abort* message to abort a subpool. A process is thrown from the subpool before the *abort* message arrives. The thrown process is delivered to a PE where there is no subpool having the same PID as the thrown process. Then a new subpool is created and a *ready* message is sent. A process may be thrown *again* to still another PE from this new subpool before the PE receives an *abort* message from the controlling process.

On receiving one *abort* message, the nonzero weight of the subpool is sent back to the controlling process. Since the sum of the weights of subpools and processes in transit is finite, all processes can be aborted by sending a finite number of *abort* messages, even if the above situation occurs.

5.3 When the Weight becomes One

As mentioned in the section 5.1, when the weight of a subpool becomes *one*, the PE cannot throw a process.

In this case, the PE sends a message requesting more weight (*request* message) to the controlling process. Process throwing is suspended until the weight of the subpool becomes more than one. On receiving a *request* message, the controlling process sends back a message which carries some weight to the sender PE (*supply* message) and reduces the same amount from its own weight. When a PE receives a *supply* message, it adds the weight carried by the *supply* message to the weight of the subpool, which enables it to throw any suspended processes. Since receiving of a thrown

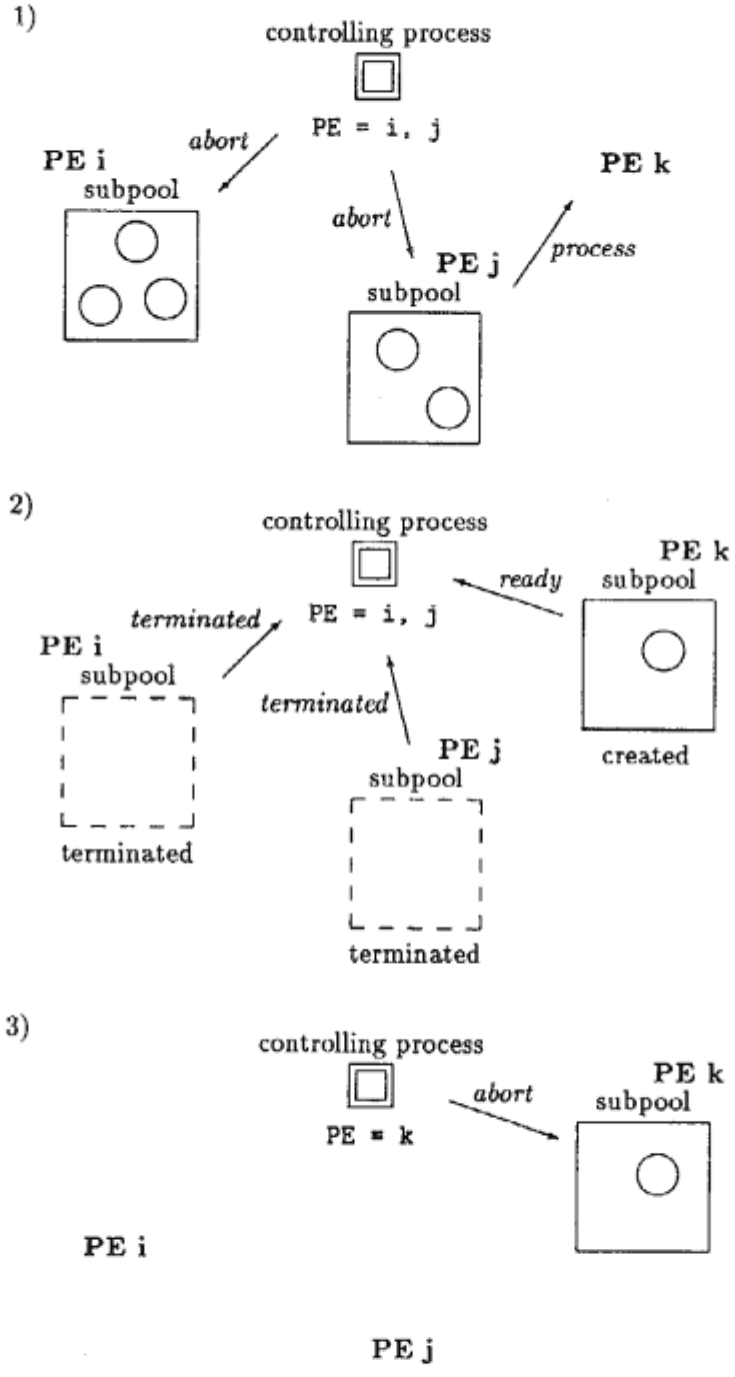

Figure 3: Abortion Operations

process also increases the weight of the subpool, a subpool may terminate before receiving a *supply* message, and a *supply* message may reach a PE that contains no subpool. In this case, a *return* message is sent back to the controlling process. This is similar to the action when a PE without a subpool receives an *abort* message.

It is not necessary to assign any weight to the *request* message, because a *terminated* message is delivered to the controlling process only after this *request* message (the channel is FIFO), and the weight of the controlling process never reaches zero, leaving *request* messages in transit.

5.4 How to Assign a Weight

This section describes the strategy to assign a weight which decreases the number of additional messages (*request* and *supply* messages).

In the worst case, that is, to assign a weight of *one* in any case, the same number of additional messages as the thrown processes are required, while no additional messages are required in the best case. If the weight carried by a *supply* message is large enough compared with the weight assigned to a thrown process, the weight of the subpool will not reach easily one after receiving a *supply* message. The weight assigned to the thrown process must be less than the weight of the subpool, while the weight carried by a *supply* message does not have this limitation. Using the following strategy, one subpool almost always needs only to send a *request* message *once*.

- Assign a fixed weight (say 2^{10}) to a thrown process if the weight of the subpool is more than twice of that; otherwise assign half of the weight of the subpool.
- A *supply* message carries a very large weight (say 2^{20}).

On receiving a *supply* message, the weight of the subpool becomes more than 2^{20} and it can throw a process at least 2^{10} times without receiving any weight.

If a subpool receives a *supply* message before its weight becomes one, it need not to send a *request* message. A subpool which is created by receiving a process with a weight of 2^{10} can throw a process at least 10 times until its weight becomes one. Therefore, if the controlling process sends back a *supply* message on receiving a *ready* message, a *request* message is expected to be almost needless.

6 Non-FIFO Communication

In the computation model with non-first-in-first-out communication, the following situations may occur:

- A *terminated* message may be delivered before a *ready* message and a *request* message.
- The controlling process may receive several *ready* messages (or *terminated* messages) before receiving a *terminated* message (or a *ready* message).

The former may cause the weight of the controlling process to reach zero, leaving *ready* messages or *request* messages in transit. On account of the latter, simply memorizing or deleting the sender PE of a *ready* message or a *terminated* message will not work. To cope with the situations mentioned above, we modify the scheme as follows:

- Assign a weight to a *ready* message and a *request* message (a *request* message will be sent when the weight reaches *two*).
- The controlling process has a set of counters corresponding to each PE, which is incremented on receiving a *ready* message and is decremented on receiving a *terminated* message.

The former change assures that the weight of the controlling process never reaches zero leaving any messages or processes in transit. By the latter change, if a subpool may exist in a PE, the value of the corresponding counter becomes positive. The controlling process thus performs the following operations to achieve the abortion:

(1) Sending an *abort* message to each PE whose corresponding count is positive;

(2) Sending an *abort* message to the sender PE of a *ready* message received after operation (1) if the count corresponding to the sender PE, after increment, is positive.

Since no more than one subpool can exist in one PE at a time, it is enough to send one *abort* message to one PE.

7 Comparison

The WTC scheme is much superior to the naive scheme using acknowledgement in two points.

First, the WTC scheme requires fewer additional messages than in the naive scheme. The number of subpools created is expected to be small enough compared with the number of thrown processes. The WTC scheme requires about the same number of *request* messages and *supply* messages as the number of the creations of subpools, while the naive scheme requires almost the same number of *acknowledge* messages as the number of thrown processes.

Second, in the WTC scheme, each subpool can terminate independently, while in the naive scheme, termination of a subpool depends on terminations of other subpools.

8 Summary

We have devised an efficient algorithm for termination detection and abortion. Its major advantages are as follows.

- Only a few additional messages are required.
- Each subpool can terminate independently.
- Reuse of the process pool identifier is possible.

The techniques described in this paper are applicable to many kinds of distributed processing systems.

Acknowledgements

We thank the members of the Multi-PSI group in the ICOT Research Center and cooperating companies, the Director of ICOT, Dr. Kazuhiro Fuchi and the manager of the fourth research laboratory, Dr. Shunichi Uchida for valuable discussions and encouragement.

References

[1] D. I. Bevan. Distributed garbage collection using reference counting. In *Proceedings of Parallel Architectures and Languages Europe*, pages 176–187, June 1987.

[2] N. Ichiyoshi, T. Miyazaki, and K. Taki. *A Distributed Implementation of Flat GHC on the Multi-PSI.* Technical Report TR-230, ICOT, 1987. Also in Proceedings of the Fourth International Conference on Logic Programming, 1987.

[3] K. Taki. The parallel software research and development tool: Multi-PSI system. In *Proceedings of France-Japan Artificial Intelligence and Computer Science Symposium 1986*, pages 365–381, 1986.

[4] K. Ueda. *Guarded Horn Clauses.* Technical Report TR-103, ICOT, 1985.

[5] 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.

[6] M. Yokota, A. Yamamoto, K. Taki, H. Nishikawa, and S. Uchida. *The Design and Implementation of a Personal Sequential Inference Machine: PSI.* ICOT Technical Report TR-045, ICOT, 1984. Also in New Generation Computing, Vol.1 No.2, 1984.