The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes

(Extended Abstract)

*Gerard Tel**
Department of Computer Science, Utrecht University,
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands.
(Email: gerard@cs.ruu.nl)

Friedemann Mattern
Department of Computer Science, Kaiserslautern University,
P.O. Box 3049, D 6750 Kaiserslautern, Fed. Rep. Germany.
(Email: mattern@informatik.uni-kl.de)

Abstract. It is shown that the termination detection problem for distributed computations can be modeled as an instance of the garbage collection problem. Consequently, algorithms for the termination detection problem are obtained by applying transformations to garbage collection algorithms. The transformation can be applied to collectors of the "mark–and–sweep" type as well as to reference counting garbage collectors. As an example, the scheme is used to transform the weighted reference counting protocol.

1 Introduction

A substantial amount of the research efforts in distributed algorithms design has been devoted to the problem of detecting when a distributed computation has terminated. There are several reasons for the impressive number of publications on this subject. First, as the problem has shown up under varying model assumptions and there are several solutions for each model, a really large number of different algorithms has emerged. All these algorithms were published separately, because unifying approaches, treating a number of algorithms as a class, have been rare. Second, the problem of termination detection, being sufficiently easy to define and yet non–trivial to solve, has been seen as a good candidate

*The work of this author was supported by the ESPRIT II Basic Research Actions Program of the EC under contract no. 3075 (project ALCOM).

to illustrate the merits of design or proof methods for distributed algorithms. Third, it has been observed that the fundamental difficulties of the termination detection problem are the same as those of other problems in distributed computing. Termination detection algorithms are related to algorithms for computing distributed snapshots [CL85], and detecting deadlocks [CMH83]. Thus the problem is seen to be important both from a practical, algorithmical, and from a theoretical, methodological point of view.

From both points of view we consider it useful to recognize general design paradigms for distributed termination detection algorithms. One such paradigm was described in [Te90]. A new paradigm is presented in this paper: it is shown that termination detection algorithms are obtained as suitable instantiations of garbage collection algorithms. A connection between the two problems was pointed out before. Tel, Tan, and Van Leeuwen [TTL88] have shown that garbage collection algorithms (of the so–called *mark–and–sweep* type, see section 1.2) can be derived from termination detection algorithms. Using a different transformation, garbage collection algorithms of the *reference counting* type can also be derived from termination detection algorithms, see section 4.1. The results in this paper further strengthen this connection by presenting a transformation in the reverse direction.

Subsections 1.1 and 1.2 introduce the termination detection problem and the distributed garbage collection problem. Section 2 describes how the termination detection problem can be formulated as garbage collecting one hypothetical object and derives the algorithmical transformation. Section 3 provides an example of the transformation. Section 4 contains some additional remarks and comments.

1.1 The Termination Detection Problem

The problem of termination detection is described formally as follows. A collection $\mathbf{P}$ of *processes* is considered, communicating by message passing. For the sake of simplicity it is assumed that $\mathbf{P}$ is a fixed collection, but the results in this paper are easily generalized to handle process creation and deletion as well. A process is either *passive* or *active*. *Active* processes can send messages, but *passive* processes cannot. An *active* process can spontaneously become *passive*, but a *passive* process can become *active* only on receipt of a message. Formally, the allowed actions of the processes are described as follows. (In all programs to follow, actions are *atomic* and braces ("{" and "}") enclose a guard for an action.)

$\mathbf{S}_p$: { $state_p = active$ }
send a message $\langle M \rangle$

$\mathbf{R}_p$: { A message has arrived }
receive message $\langle M \rangle$; $state_p := active$

$\mathbf{I}_p$: { $state_p = active$ }
$state_p := passive$

Define the *termination condition* as:

No process is *active* and no messages are in transit.

When processes behave as described, this condition is stable: once true, it remains so. The problem of termination detection now is to superimpose on the described *basic* computation a *control* computation which enables one or more of the processes to detect when the termination condition holds. To this end a new special state *terminated* is introduced for each process. The following two criteria specify the correctness of the control algorithm.

T1 *Safety.* If some process is in the *terminated* state then the termination condition holds.

T2 *Liveness.* If the termination condition holds, then eventually some process will be in the *terminated* state.

A *passive* process may take part in this control computation, and receiving control messages does not make a *passive* process *active*.

Solutions to the termination detection problem are non–trivial, mainly due to the possibility that a process becomes *active* after being observed as *passive* by the control algorithm. Several classes of solutions are known. The most important ones are those based on *probes* and those based on *acknowledgements*. The best known example of the former class is [DFG83], and a general treatment is given in [Te90]. Examples of the latter class are [DS80, SF86]. Solutions based on counting sent and received messages are proposed in [Ma87].

1.2 The Distributed Garbage Collection Problem

As our approach for deriving termination detection algorithms is based on solutions to the garbage collection problem, we shall now describe this problem in a model which is close to the model of Lermen and Maurer [LM86]. The advantage of this model is that it abstracts from aspects which are not relevant to our purposes, such as processors, memory cells, and the difference between "local" and "remote" references.

An (object–oriented) distributed system consists of a collection O of cooperating *objects*. A subset of O is designated as *root objects*. Objects are able to hold *references* to other objects. These references can be transmitted in messages, see below. A reference to an object r will be called an r–reference. An object r is a *descendant* of q if q holds an r–reference or a message containing an r–reference is in transit to q. An object is *reachable* if it is a root object or a descendant of a reachable object. An object p holding an r–reference may *delete* it, after which p no longer holds this reference. Also, a reachable object p holding an r–reference may *copy* the reference to another object q, by sending an r–reference in a message to q. Object q will hold an r–reference after receipt of this message. An object can have multiple references to the same target object.

An object is called *garbage* if it is not reachable. As only reachable objects copy references, only references to reachable objects are copied, and thus a garbage object remains garbage forever. For reasons of memory management it is required that garbage objects are identified and collected. This task is taken care of by a garbage collecting algorithm. The following two criteria define the correctness of a garbage collecting algorithm.

G1 *Safety.* If an object is collected, it is garbage.

G2 *Liveness.* If an object is garbage, it will eventually be collected.

Many solutions have been proposed to the distributed garbage collection problem, most of which fall into one of two categories: collectors of the *reference counting* type and collectors of the *mark–and–sweep* type. Both types of solutions have been known for over 30 years for classical, non–distributed systems [Co60, McC60].

Collectors of the first type [LM86, WW87, Be89] maintain for each non–root object a count of the number of references in existence to that object. References in other objects as well as references in messages are taken into account. The reference count is incremented when a corresponding reference is copied, and decremented when such a reference is deleted. When the count for an object drops to zero, it can be concluded that the object is garbage and consequently the object can be collected. Reference counting garbage collectors are unable to collect *cyclic garbage* (a collection of garbage objects pointing to each other). As will be seen at the end of section 2, this does not render our transformation invalid for reference counting garbage collectors.

Collectors of the second type [Dij78] mark all reachable objects as such, starting from the roots and recursively marking all descendants of marked objects. In this way all reachable objects become marked eventually. The design of the marking algorithm is complicated by the possibility that references are copied and deleted during its operation. The objects must cooperate with the marking algorithm, e.g., by also marking objects

when references are copied, cf. [Dij78]. When the marking phase is terminated a sweep through all objects is made, in which all unmarked objects are collected. These two phases are repeated as long as necessary.

2 Termination Detection Using Garbage Collection

In this section we describe how the termination detection problem in general can be modeled as an instance of the garbage collection problem. As a result, solutions to the termination detection problem can be derived from garbage collection algorithms, of which an example will be shown in section 3. First the collection O of objects used for this purpose is described, as well as the behavior of these objects. Next it is shown that the termination condition is equivalent to one particular object becoming garbage, so termination can be detected by a garbage collection algorithm.

Recall that P is the set of processes whose termination is to be detected. The collection O of objects consists of one root object A_p for every process p in P, and a single *indicator object* Z. Object A_p mimics the behavior of process p as far as the basic computation is concerned (it sends and receives p's basic messages, and has all the variables p has). Messages may contain a reference, in which case the message is a copy message for that reference. Object A_p is called *passive* (*active*) when the mimicked process p is *passive* (*active*). As A_p is a root object, it is always reachable.

The indicator object Z is not a root object. Its only purpose is to indicate the termination condition with its reachability status by the following equivalence, which will be maintained during execution.

(IND) $\qquad Z$ is garbage $\Leftrightarrow$ the termination condition holds.

Theorem 2.1 *IND holds when the following two rules are observed:*

R1 *An object holds a Z–reference if and only if it is active.*

R2 *Each message of the basic computation contains a Z–reference.*

Proof. Z is garbage is equivalent to: Z is not a descendant of any of the A_p. By definition, this means that no A_p holds a Z–reference, and to no A_p a message is in transit containing a Z–reference. By R1 and R2 this is equivalent to: no A_p is *active* and to no A_p a message (of the basic computation) is in transit. This is the definition of the termination condition. □

It remains to be shown how R1 and R2 can be maintained. It is possible to ensure through proper initialization that R1 and R2 hold initially. To this end, assume that *active* objects are initialized with the necessary Z–reference, and *passive* objects without it, and that messages in transit initially contain the reference also. To maintain R1 and R2 during the distributed computation, each transmission of a message copies the Z–reference, and processes delete their Z–reference when they become *passive*. More explicitly, the actions to be carried out by A_p are modified as follows:

$\mathbf{S}_p$: { $state_p = active$ }
send a message $\langle M, Z \rangle$

$\mathbf{R}_p$: { A basic message has arrived }
receive message $\langle M, Z \rangle$; $state_p := active$;
insert Z in the references of A_p

$\mathbf{I}_p$: { $state_p = active$ }
$state_p := passive$;
delete Z from the references of A_p

With these modifications R1 and R2 are maintained indeed. R1 is maintained because Z–references are deleted in action $\mathbf{I}_p$, and inserted in action $\mathbf{R}_p$. The latter is possible because the message contains a Z–reference by R2. R2 is maintained because in action $\mathbf{S}_p$ a Z-reference is included in every message. This is possible because only *active* objects send messages, and these objects contain a Z–reference by R1. Thus R1 and R2 are maintained during computation, and by theorem 2.1 IND holds. To arrive at a termination detection algorithm, superimpose upon the objects as described a garbage collection algorithm to detect that Z is garbage. The garbage collection algorithm is then modified so as to inform the objects A_p when Z is identified as garbage. (On receiving this notice, the root objects enter the *terminated* state. We omit this (trivial) operation from the description of the algorithms that will follow.)

Theorem 2.2 *The algorithm as constructed satisfies conditions T1 and T2.*

Proof. Assume any process enters the *terminated* state. This happens upon notice that Z is collected. By the correctness of the garbage collection algorithm (condition G1) this implies that Z is garbage. By IND the termination condition holds.

Assume the termination condition holds. By IND, Z is garbage, hence, by the liveness of the garbage collector (condition G2) Z will eventually be collected. Notice of this will be sent to the processes, and these will enter the *terminated* state in finite time. □

Garbage collectors of the reference counting type are not able to collect cyclic structures of garbage, which may possibly harm the liveness of the termination detection algorithm. It is, however, easily seen that Z is not part of such a cyclic structure, and in fact the following, stronger equivalence holds.

There are no references to $Z \Leftrightarrow$ the termination condition holds.

Summary of the transformation. The construction of a termination detection algorithm is summarized in the following four steps.

1. Form the set **O** of objects, consisting of the root objects A_p and one indicator object Z.
2. Superimpose upon the actions of the basic computation the handling of the Z–reference.
3. Superimpose upon this combined algorithm a garbage collection algorithm.
4. Replace the collection of Z (or its identification as garbage) by a notification of termination.

3 An Example of the Transformation

The transformation described in section 2 can in principle be applied to any garbage collection scheme, of the reference counting as well as the mark–and–sweep type, or working according to other principles. In this section we consider the transformation of a garbage collection algorithm based on weighted reference counting. The resulting termination detection algorithm turns out to be an already known algorithm: it was proposed in [Ma89]. More derivations, yielding new and non–trivial termination detection algorithms, are found in the full paper [TM90].

In a weighted reference counting scheme, each reference has an associated positive weight. Each object o maintains a reference count, which equals (barring certain update delays) the total weight of existing o–references. (The term "reference weight accumulator" might be more appropriate for this variable, but in accordance with the existing literature we shall continue to use the word "count".) When a reference is copied, its weight is *split* among the existing and the new reference. Thus, although the *number* of references increases, the *weight* remains the same, and the reference count need not be incremented and no message need be sent to the referenced object. When an object deletes an r–reference, a decrement message is sent to r, reporting the weight of the deleted

reference. Upon receipt of this message, r subtracts this weight from its reference count (and is collected if the count drops to zero).

3.1 Description of the Scheme

Distributed weighted reference counting schemes have been given by Watson and Watson [WW87], Bevan [Be89], and others. In the description below the mechanism to create new objects is omitted, because in the transformation no new objects are ever created. An o–reference is a tuple (o, w), where w denotes the weight of the reference. Initially for each non–root object o, the reference count RC_o equals the sum of the weights of all existing o–references. The following (atomic) actions can take place. ($\mathbf{CR}_p$ represents the sending of a copy message, $\mathbf{RR}_p$ the receipt of such a message, $\mathbf{DR}_p$ the deletion of a reference and the associated sending of a decrement–weight message, and $\mathbf{RD}_o$ the receipt of such a message.)

$\mathbf{CR}_p$: { p holds reference (o, w) }
send $\mathbf{cop}(o, w/2)$ to q ; $w := w/2$

$\mathbf{RR}_p$: { A message $\mathbf{cop}(o, w)$ has arrived at p }
receive $\mathbf{cop}(o, w)$;
if p has an o–reference
then add w to its weight
else insert the o–reference with weight w

$\mathbf{DR}_p$: { p holds reference (o, w) }
send $\mathbf{dec}(o, w)$ to o ; delete the o–reference

$\mathbf{RD}_o$: { A $\mathbf{dec}(o, w)$ message has arrived at o }
receive $\mathbf{dec}(o, w)$; $RC_o := RC_o - w$;
if $RC_o = 0$ **then** collect o

Action $\mathbf{RR}_p$ guarantees that in this scheme an object has at most one reference to each other object. A correctness proof and analysis of the scheme is found in [WW87] or [Be89] and is based on invariance of the following two assertions:

1. Each reference has a positive weight; each delete message contains a positive weight.
2. $RC_o = \sum_{R=(o,w)} w + \sum_{D=\mathbf{dec}(o,w)} w$, where R ranges over all o–references in existence (including **cop** messages) and D ranges over all delete messages in transit.

3.2 Transformation into a Termination Detection Algorithm

To transform the garbage collection scheme into a termination detection algorithm we apply the four–step construction of section 2.

1. The set **O** of objects consists of the objects A_p and the indicator object Z.

2. Superimpose upon the actions of the basic computation the handling of the Z–reference. This yields the following program text.

   $\mathbf{S}_p$: { $state_p = active$ }
   send a message $\langle M, Z \rangle$

   $\mathbf{R}_p$: { A basic message has arrived }
   receive message $\langle M, Z \rangle$; $state_p := active$;
   insert Z in the references of A_p

   $\mathbf{I}_p$: { $state_p = active$ }
   $state_p := passive$;
   delete Z from the references of A_p

3. Superimpose the reference counting scheme upon these actions. To this end, action $\mathbf{CR}_p$ is included in action $\mathbf{S}_p$, action $\mathbf{RR}_p$ is included in action $\mathbf{R}_p$, and action $\mathbf{DR}_p$ is included in action $\mathbf{I}_p$. For o the object Z is substituted. This results in the following program text.

   $\mathbf{S}_p$: { $state_p = active$ and p holds reference (Z, w) }
   send a message $\langle M, \mathbf{cop}(Z, w/2) \rangle$; $w := w/2$

   $\mathbf{R}_p$: { A basic message has arrived }
   receive message $\langle M, \mathbf{cop}(Z, w) \rangle$; $state_p := active$;
   if p has a Z–reference
   then add w to its weight
   else insert the Z–reference with weight w

   $\mathbf{I}_p$: { $state_p = active$ }
   $state_p := passive$;
   send $\mathbf{dec}(Z, w)$ to Z ; delete the Z–reference

RD$_Z$: { A **dec**(Z, w) message has arrived at Z }
receive **dec**(Z, w) ; $RC_Z := RC_Z - w$;
if $RC_Z = 0$ **then** collect Z

4. Replace the collection of Z by a notification of termination. Some more simplifications can be made in addition: the actual handling of the Z reference can be removed; instead we equip every process p with a variable W_p, representing the weight of p's (virtual) Z–reference (0 if p has no such reference). The subscript Z is dropped. This finally results in the following algorithm.

S$_p$: { $state_p = active$ }
send a message $\langle M, W_p/2 \rangle$; $W_p := W_p/2$

R$_p$: { A basic message has arrived }
receive message $\langle M, W \rangle$; $state_p := active$;
$W_p := W_p + W$

I$_p$: { $state_p = active$ }
$state_p := passive$;
send **dec**(W_p) to Z ; $W_p := 0$

RD: { A **dec**(W) message has arrived at Z }
receive **dec**(W) ; $RC := RC - W$;
if $RC = 0$ **then** send **term** to all A_p

The initial conditions for this algorithm are: $W_p = 0$ if p is *passive*; $W_p > 0$ if p is *active*; $RC = \sum_p W_p$; and no messages are in transit. (Or, if there are messages, RC correctly reflects their weight.)

The termination detection algorithm that has just been derived is known as the *Credit Recovery* algorithm [Ma89]. The algorithms discussed in this section face the problem of so–called *weight underflow*. When weights are represented in a finite number of bits, there exists a smallest positive value a weight can take, and it is not possible to split this weight in two positive parts. Furthermore, the accumulation of small fragments may cause problems. Solutions to these problems and variants of the scheme may be found in [Be89, Ma89].

4 Conclusions

In this paper we have presented a transformation of garbage collection schemes into termination detection algorithms. Applying the transformation to the weighted reference counting scheme, we have derived the Credit Recovery algorithm for termination detection. Virtually all garbage collection schemes can be transformed into sensible termination detection algorithms. The full paper [TM90] contains derivations of more termination detection algorithms, including three new ones: the Activity Counting algorithm, the Generational termination detection algorithm, and a "dual-tour" token algorithm for a ring of processes. It also contains a discussion of several related aspects, of which we only sketch two here.

4.1 Reverse Transformation

It is also possible to transform a termination detection algorithm into a reference counting garbage collection scheme. The aim of a reference counting algorithm is to collect an object o when all o–references (in objects) have been deleted and no more o–references are in transit (in copy messages).

An object is defined to be *o–active* if it holds an o–reference and *o–passive* otherwise, and a message is called an o–activation message if it carries an o–reference. Under these definitions, an *o–passive* object becomes *o–active* only upon receipt of an o–activation message, and only *o–active* objects send o–activation messages. Now the *o–termination condition*, defined as:

No process is *o–active* and no o–activation messages are in transit

is stable and can be detected by a termination detection algorithm. Furthermore

(RT) There are no o–references $\Leftrightarrow$ the o–termination condition holds.

To arrive at a reference counting garbage collection algorithm, a termination detection algorithm is superimposed on the o–reference handling. When o–termination is detected, o is collected. For each object a separate instance of the termination detection algorithm is executed concurrently.

Although this transformation could be applied to any termination detection algorithm, the resulting reference counting garbage collection scheme would not be feasible in all cases. A complete algorithm along these lines, based on the algorithm in [DS80], was proposed by Rudalics [Ru90].

4.2 Deadlock Detection

The termination detection problem is an instance of a class of detection problems in distributed systems. Communication deadlock detection is a generalization where also a part of the network can be terminated. In this problem, for each *passive* process a subset of the processes is determined at the moment it becomes *passive*. The process can become *active* only by receiving a message from a process in this subset. The termination detection problem is obtained, when each process always chooses the full set of processes. We are currently investigating how the approach in this paper can be generalized to derive (mark–and–sweep) deadlock detection algorithms from garbage collection algorithms.

Acknowledgements: We want to thank Martin Rudalics and Jörg Richter for their discussions of the paper and numerous suggestions and comments. We want to thank the Eindhoven Tuesday Afternoon Club and Reinhard Schwarz for their careful revision of the text.

References

[Be89] Bevan, D.I., *An Efficient Reference Counting Solution to the Distributed Garbage Collection Problem,* Parallel Computing 9 (1989) 179–192.

[CL85] Chandy, K.M., L. Lamport, *Distributed Snapshots: Determining Global States of Distributed Systems*, ACM Trans. on Computer Systems 3 (1985) 45–56.

[CMH83] Chandy, K.M., J. Misra, L.M. Haas, *Distributed Deadlock Detection*, ACM Trans. on Computer Systems 1 (1983) 144–156.

[Co60] Collins, G.E., *A Method for Overlapping and Erasure of Lists*, Comm. ACM 3 (1960) 655–657.

[DFG83] Dijkstra, E.W., W.H.J. Feijen, A.J.M. van Gasteren, *Derivation of a Termination Detection Algorithm for Distributed Computations,* Inf. Proc. Lett. 16 (1983) 217–219.

[Dij78] Dijkstra, E.W., L. Lamport, A.J. Martin, C.S. Scholten, E.F.M. Steffens, *On–the–fly Garbage Collection: An Exercise in Cooperation*, Comm. ACM 21 (1978) 966–975.

[DS80] Dijkstra, E.W., C.S. Scholten, *Termination Detection for Diffusing Computations*, Inf. Proc. Lett. 11 (1980) 1–4.

[LM86] Lermen, C.–W., D. Maurer, *A Protocol for Distributed Reference Counting*, ACM Conference on Lisp and Functional Programming, Cambridge, 1986, pp. 343–354.

[Ma87] Mattern, F., *Algorithms for Distributed Termination Detection*, Distributed Computing 2 (1987) 161–175.

[Ma89] Mattern, F., *Global Quiescence Detection Based on Credit Distribution and Recovery*, Inf. Proc. Lett. 30 (1989) 195–200.

[McC60] McCarthy, J., *Recursive Functions of Symbolic Expressions and Their Computation by Machine*, Comm. ACM 3 (1960) 184–195.

[Ru90] Rudalics, M., *Implementation of Distributed Reference Counts*, Technical Report (forthcoming), Research Institute for Symbolic Computation, J. Kepler University, Linz, 1990.

[SF86] Shavit, N., N. Francez, *A New Approach to Detection of Locally Indicative Stability*, in: L. Kott (ed.), *Proceedings ICALP 1986*, Lecture Notes in Computer Science 226, Springer–Verlag, 1986, pp. 344–358.

[Te90] Tel, G., *Total Algorithms*, Technical Report RUU–CS–88–16, Dept. of Computer Science, Utrecht University, 1988. Also in: Algorithms Review 1 (1990) 13–42.

[TM90] Tel, G., F. Mattern, *The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes*, Technical Report RUU–CS–90–24, Dept. of Computer Science, Utrecht University, 1990.

[TTL88] Tel, G., R.B. Tan, J. van Leeuwen, *The Derivation of Graph Marking Algorithms from Distributed Termination Detection Protocols*, Science of Computer Programming 10 (1988) 107–137.

[WW87] Watson, P., I. Watson, *An Efficient Garbage Collection Scheme for Parallel Computer Architectures*, in: J.W. de Bakker, A.J. Nijman, P.C. Treleaven (eds.), Proceedings Parallel Architectures and Languages Europe, vol. II, Lecture Notes in Computer Science 259, Springer–Verlag, 1987, pp. 432–443.