Algorithmica (1990) 5: 325-340

Algorithmica

Asynchronous Distributed Termination—Parallel and Symmetric Solutions with Echo Algorithms[1]

Friedemann Mattern[2]

Abstract. The principle of message counting is used to detect termination of distributed computations which consist of processes asynchronously communicating over non-FIFO channels. The solution is symmetric and not based on a predefined communication structure. An efficient variant of the echo algorithm, which dynamically builds a spanning tree, allows a parallel and distributed evaluation of the termination predicate in time proportional to the diameter of the communication graph. Concurrent and repeated initiation of the detection algorithm by different processes is possible at any time without prior synchronization due to a subtle method of collision detection and wave extinction, which can be regarded as a distributed election scheme where the average message complexity increases only logarithmically with the number of concurrent initiators. Control messages have a small length and additional communication links are not required. Only a fixed number of simple variables is needed in every process, global knowledge such as the total number of processes or the structure of the network is not used, making the scheme useful for dynamic systems. Several variations of the basic principle are presented, important issues such as message complexity and fault-tolerance are discussed.

Key Words. Distributed termination, Echo algorithm, Distributed programming, Decentralized control, Distributed algorithm, Election, Symmetry.

1. Introduction. Determining when the computation of a collection of communicating processes has terminated is a fundamental and nontrivial problem in distributed programming, and in recent years the problem of *distributed termination detection* has received much attention (see among others [9], [11], [10], [5], and [18]). Although the problem is simple to formulate, a surprising variety of algorithms with rather different properties have been published. Whereas most solutions are based on a synchronous model of communication (notably CSP), we present and discuss solutions for a more general model, where message passing is *asynchronous* with *arbitrary but finite communication delays.* Due to the fact that global time [17] or a consistent snapshot [4] of a distributed system is *a priori* not available, the main problem which consists in detecting messages that are in transit, is nontrivial. As in [15], [16], and [19], the algorithms we present here are based on the principle of message counting.

In order to visit every process and collect useful information we make use of a variant [22] of the *echo algorithm* [8]. An *initiator* suspecting termination sends

[1] This work was supported by the German National Science Foundation (Deutsche Forschungsgemeinschaft) as part of research project SFB124.

[2] Department of Computer Science, University of Kaiserslautern, P.O. Box 3049, D-6750 Kaiserslautern, Federal Republic of Germany.

Received November 24, 1987; revised April 15, 1988. Communicated by C. K. Wong.

out *query messages* to all neighboring processes. Upon receipt of the first query message a node becomes *engaged* and propagates the "query wave" further on to all its neighbors. A node having no other communication link than the one from which it has received the query message, immediately returns an *echo* which contains among other things the value of a counter indicating the number of sent minus the number of received messages of the underlying computation by that node. Having received queries or echoes along every incident communication link, a node becomes disengaged and returns an echo with the accumulated values to the node from which it was engaged. Eventually the "echo wave" reaches the initiator.

Two problems emerging from this principle have to be solved: the first one is concerned with possible *inconsistencies* due to the fact that the nodes are visited at different times, possibly leading to false termination detection. We will prove that the system is actually terminated when the accumulated value of the message counters at the initiator is 0 and no process node received a basic message while it was engaged. This property can easily be implemented by a flag which is set on receipt of a basic message, reset by the query wave, and tested and accumulated by the echo wave. The second problem is concerned with *contemporary invocations* of the algorithm by different processes: whereas in principle every node can have its own set of control variables at every other node (possibly allocated dynamically when required) so that "control waves" initiated by different nodes would not interfere, mainly for reasons of economy we prefer solutions where a node is engaged in at most one wave at a time. Because we do not want the nodes to synchronize themselves for exclusive invocation of the algorithm, collisions of control waves must be detected and handled in an appropriate way. In particular, the possibility of deadlock or starvation must be avoided by still keeping the scheme as "symmetric" as possible.

Besides being simple and readily implementable, the symmetric solution presented in this paper has a number of interesting properties. Control messages have a small length and additional communication links are not required. Communication can be synchronous or asynchronous and messages can be received out of order. Every node can repeatedly start the detection algorithm independently of other nodes, prior synchronization is not required. The detection algorithm itself is distributed, the termination condition is evaluated in parallel. No node needs global knowledge such as the structure of the network or the total number of nodes. Symmetry, independence of topology, and absence of global knowledge and centralized control promote robust and fault-tolerant variants.

The paper is organized as follows: in Section 2 we present our model of distributed computation and define the distributed termination problem. In Section 3 we describe the general principle of the algorithm, assuming that different executions are mutually exclusive. Its correctness is proven in Section 4. Section 5 describes concurrent initiations and the principle of wave extinction. In Section 6 we present another, similar algorithm; the two solutions and several variants are discussed in Section 7. In Section 8 we compare our algorithms to related solutions. Finally, we summarize the conclusions in Section 9.

2. The Model. For the application of the algorithms we consider static distributed systems with at least two processes having distinct totally ordered identities different from 0. The processes (also called nodes) are connected by *bidirectional* communication links forming a *connected* communication graph. We assume that all messages sent arrive within some finite undetermined time (message losses will be discussed in Section 7); however, we do *not* require the FIFO property. The messages of the underlying application are called *basic messages*, whereas messages of the superimposed termination detection algorithm (i.e., *queries* and *echoes*) make up the *control communication.*

With respect to the underlying computation a process is in one of two states, *active* or *passive.* The system behaves according to the following rules:

(1) Only active processes may send basic messages.
(2) A process may change from active to passive at any time.
(3) A process may change from passive to active only on receipt of a basic message.

The underlying computation has *terminated* when all processes are passive and no basic messages are in transit. The *problem* is to devise an algorithm which detects this persistent property of the global state by means of extra control communication. In order to superimpose a termination detection algorithm on the underlying computation properly, we assume that control messages are only accepted when a process is passive and that no other messages are accepted while it is processing a control message, i.e., the actions of the message-driven control computation are supposed to be *atomic.* We further assume that a process initiates termination detection only when it is passive.

3. The First Algorithm—Distributed Time Priority. In order to simplify the description we assume for the moment that different executions of the algorithm are mutually exclusive. The principle of the method is simple: an initiator which is passive with respect to the underlying computation may start an execution of the echo algorithm. The query wave then spreads "down" resetting a communication flag COM in every node. The echo wave accumulates the local message counters S and the communication flag. If the initiator detects that the total number of registered messages sent and received are equal and that the flag was not set due to the receipt of a basic message while a node was engaged, it signals termination.

We first explain the task of the variables and their initializations (if relevant) used in the realization of the algorithm. A process node is unaware of the global structure of the network, it only needs to know the links leading to its neighboring nodes. However, for the sake of easy readability we assume that the identities of the neighboring nodes are stored in the set NEIGHBORS. A counter N is used to count the incoming control messages (line 17). A variable PRED is used to hold the identity of the preceding node (line 4) from which the first control message has been received (PRED = 0 for the initiator) so that when all control

messages have been received (line 18) an echo can be sent back (line 19). We further assume that a boolean flag COM (reset by the query wave in line 7) is set to true on receipt of a basic message (line 24) and that a counter S (initialized to 0 prior to the start of the basic computation) is incremented whenever a basic message is sent and decremented if one is received (lines 24 and 25). The variable ACCU serves as an accumulator for S-values received by echoes (line 13). The CLOCK is initialized to some minimal value, its exact function will be explained in Section 5 when we are concerned with concurrent executions of the algorithm. Together with the variable T its function is among others to discern primary queries from secondary queries which may arrive later on other edges instead of an echo (line 10).

```
Upon receiving QUERY(T) from process p:
 1. if T > CLOCK then   /* primary query */
 2.    CLOCK ← T;   /* remember the new time */
 3.    if PRED = 0 and N ≠ |NEIGHBORS| then failed fi;   /* extinction */
 4.    PRED ← p;
 5.    if |NEIGHBORS| = 1 then send ECHO(T, S, false) to p;   /* terminal */
 6.    else N ← 1;   /* initializations */
 7.       START: ACCU ← 0; COM ← false;
 8.       send QUERY(CLOCK) to NEIGHBORS − {PRED};   /* propagate
             queries */
 9.    fi;
10. elseif T = CLOCK then CHECK_COMPLETED;   /* secondary queries */
11. fi;

Upon receiving ECHO(T, SUM, FAILED):
12. if T = CLOCK then   /* not an old wave */
13.    ACCU ← ACCU + SUM;   /* accumulate */
14.    COM ← COM or FAILED;
15.    CHECK_COMPLETED;
16. fi;

procedure CHECK_COMPLETED:
17.  N ← N + 1;   /* count control messages */
18.  if N = |NEIGHBORS| then   /* complete */
19.     if PRED ≠ 0 then send ECHO(T, ACCU + S, COM) to PRED;
20.     elseif ACCU + S = 0 and not COM then terminated;
21.     else failed;
22.     fi;
23.  fi;
end CHECK_COMPLETED;

When receiving a basic message:
24.  S ← S − 1; COM ← true;

When sending a basic message:
25.  S ← S + 1;
```

The initiator *starts* the algorithm by setting N and PRED to 0, advancing the CLOCK, and beginning execution at the label START (line 7). Query messages are sent to all neighbors (line 8). The initiating actions can be stated explicitly as follows:

```
N ← 0; PRED ← 0; ACCU ← 0; COM ← false;
CLOCK.t ← CLOCK.t + 1;   /* see first paragraph */
CLOCK.i ← this_process_id;   /* of Section 5 below */
send QUERY(CLOCK) to NEIGHBORS;
```

When a query is received for the first time, its parameter T is guaranteed to be greater than the local CLOCK-value (line 1). As a consequence, either the query is propagated (line 8) or, if it reached a terminal node, an echo is generated (line 5). Subsequent receipts of secondary query messages are merely counted (lines 10 and 17). Echoes are also counted (lines 15 and 17) after the values sent with them have been accumulated (lines 13 and 14). When the echo wave reaches the initiator (PRED = 0) a test is made to check whether the accumulated message counter equals 0 and no node recorded the receipt of a basic message while it was engaged (line 20). If that is the case, termination is reported. Otherwise (line 21) the algorithm can be restarted at some later time, repeated executions of the algorithm pose no problems.

By changing the statement "COM ← **true**" to "ACCU ← ACCU + 1" in line 24 it is possible to dispense with the boolean flag COM (and the last parameter of echo messages). Because this compensates the effect of "$S \leftarrow S - 1$" in the same line, this can be regarded as if basic messages being received while a node is engaged are not registered. However, the correctness proof of the algorithm (Section 4) is simplified by keeping the communication flag.

Lines 3 and 12, the parameter T of echo messages, the second component of CLOCK, and proper initializations of PRED and N prior to the start of the basic computation at all process nodes are only relevant when considering concurrent invocations of the algorithm (Section 5).

4. Correctness of the Method. The correctness of the method is based on the following propositions:

(a) *The echo principle is correctly realized by the algorithm (i.e., the initiator receives the accumulated values of all nodes after finite time).*

(b) *If the system was already terminated at the start of a detection round, termination will be detected at the end of the round.*

(c) *If at the end of the detection algorithm the accumulated communication flag is not set and the accumulated message counter equals* 0, *the system is truly terminated.*

We dispense with a proof of (a) since the echo principle as a scheme for collecting information from the nodes of a network is discussed in detail in [8] and [22]. Note that to guarantee termination of the echo algorithm we must

assume that no process remains in its active state forever. Property (b) is simple to realize: since all basic messages that were sent were also received, the accumulated message counter is 0 and the flag is not set during execution of the detection algorithm. Hence, if the application eventually terminates and the algorithm is repeatedly executed, termination will finally be detected in line 20.

Safety property (c) asserts that no false termination is reported by the algorithm. This is the most crucial point in the proof of the algorithm, it guarantees that the criterion used in line 20 is correct. In order to motivate the proof, the following "colorful" description of the echo scheme may be helpful, though it is not essential for the proof.

Assume that originally all nodes and edges of the communication graph are white and the initiator turns red upon the start of the algorithm. Query messages are red, echoes green. A message colors edges (along which it travels) in its own color. A red message arriving at a white node colors that node red. A (red) node having received red or green messages along *all* its incident edges becomes green (before it possibly sends an echo). Clearly, every node changes from white via red to green, for terminal nodes the red phase is rather short. It is easy to see that after the execution of the algorithm the green edges (which were first colored red by a message moving in one direction and later green by an opposite echo message) build a *spanning tree* of the communication graph. Edges remaining red were concurrently colored by two red messages moving in opposite directions. Since each edge is used by exactly two messages, this variant of the echo algorithm is also useful to construct dynamically a spanning tree with exactly $2e$ messages, where e denotes the number of edges of the underlying graph.

A sequence of *technical lemmata* will now prepare the proof of assertion (c). In accordance with the above description, the process states are designated by colors with the following meaning:

White: not yet received a control message.
Red: query message received, but echo message not yet sent ("engaged").
Green: all control messages received, echo message sent.

(1) *If a basic message is received by a red process, its local communication flag is set.*
PROOF: Line 24 of the algorithm (which is independent of the color of the process).

(2) *If a basic message is sent by a red process, the communication flag of the initiator is set.*
PROOF: The process was passive after turning red. In order to become active to send a basic message it must have received an activating basic message. According to (1) this sets its flag. The flag is not reset in the current execution of the algorithm and its setting is conveyed to the initiator by the echo wave when the process eventually becomes green.

(3) *A white process cannot receive a basic message sent by a green process.*
PROOF: All neighbors of a green process are colored (i.e., they are red or green) because a process becomes green only after having received control messages from all neighbors. Only colored processes send control messages

and colored processes do not change back to white. Hence, basic messages sent by green processes are received by colored processes only.

(4) *If at the end of the detection algorithm the communication flag is not set, all basic messages received by a white process have also been sent by a white process.*
PROOF: Follows directly from (2) and (3).

(5) *A basic message sent and received by white processes does not change the accumulated message counter.*
PROOF: Its sending as well as its receipt is registered.

(6) *If at the end of the detection algorithm the system is not terminated and the flag is not set, there exists a basic message sent by a white process which was or will be received by a green process.*
PROOF: At the end of the detection algorithm all processes are green. From the definition of termination it follows that a basic message is still in transit to a green process or a green process is active if the system is not terminated. In the latter case the process was activated by a basic message while it was already green (processes do not change colors while they are active). Because green processes are not activated spontaneously, there must exist a green process receiving an activating basic message from a nongreen process. Because by hypothesis the flag is not set, the sender is not red (2). Hence, it must be white.

(7) *If a basic message sent by a white process is received or will be received by a green process and the flag is not set, the accumulated counter is greater than* 0.
PROOF: The sending of the message is noticed, but no message receipt is registered: no red process receives a message if the flag is not set (1) and messages received by white processes do not affect the counter (4), (5). Messages received by green processes arrive too late to be registered.

(8) *If the system is not terminated at the end of the algorithm, then the flag is set or the accumulated counter is greater than* 0.
PROOF: Follows directly from (6) and (7).

This completes the proof of the algorithm, since (8) is already the proof of assertion (c).

5. Concurrent and Independent Initiations. In order to allow concurrent activations of the detection algorithm to take place without the necessity of prior synchronization, we make use of the principle of *wave extinction* together with a simple virtual time mechanism. A wave is tagged with its starting time and a newer wave will extinguish an older wave. *Virtual time* is considered to be a set of integer pairs (t, i) which are linearly ordered by the definition $(t, i) > (t', i') \Leftrightarrow t > t'$ or $t = t'$ and $i > i'$. The CLOCK is initialized to $(0, 0)$. When a new wave is started, the initiating node advances the CLOCK by setting CLOCK.t to CLOCK.$t+1$ and CLOCK.i to its process identification number. This guarantees that the starting times of different waves are different.

Lines 1, 10, and 12 guarantee that an old wave ($T <$ CLOCK) is absorbed without any consequences when it meets a newer wave. Because time is linearly

ordered it is not possible that two or more waves mutually extinguish themselves. Furthermore, *an initiator is guaranteed to be informed of the extinction of its wave*: whenever a new query message arrives at a node this node is checked (line 3) whether it is still engaged ($N \neq$ |NEIGHBORS|) in a wave it has started (PRED = 0). For that purpose, N must be initialized to the number of neighbors and PRED to some value different from 0 at every node. It is easy to see that whenever two different waves are concurrently active, the older initiator will—sooner or later—receive a query message with a higher timestamp. (This is not necessarily the timestamp of the other initiator since an even newer wave could have emerged in the meantime!)

The solution is *symmetric* in that every node has a fair chance of completing the execution without being extinguished. The priorities based on node identifications are only used for arbitration purposes in the case of a collision of two waves with identical "local" times. To minimize the possibility of collisions the termination test should only be started if the node is not engaged in the computation of another execution of the algorithm, i.e., if $N =$ |NEIGHBORS|.

Although it is not possible that several concurrently active waves mutually extinguish themselves, a *dynamic blocking situation* could emerge if nodes restart the algorithm too early: a node, knowing about the recent extinction of its wave, could as soon as it is disengaged by its "opponent" (line 19 or line 5) restart the algorithm with a higher timestamp and chase after its opponents wave to take "revenge." If two or more nodes repeatedly behave in that way, no node would succeed in successfully terminating the test. The situation is reminiscent of the collision problem in CSMA/CD local-area networks, and in fact the solutions of using timeouts to reduce the frequency of initiating the termination test could also be applied here. Whether this is a practicable method or whether the optimistic approach ("collisions will rarely occur") should be abandoned in favor of a pessimistic approach (synchronization for mutual exclusion or election of a starter-process) is a pragmatic question we will not endeavor to discuss here. In any case, the dynamic blocking problem is solved by the variant presented in the next section.

The starvation problem can also be avoided if a node whose wave has been extinguished never starts a new wave again. Since the waves are linearly ordered, at least one wave will run to completion. The initiator of that wave can restart the detection algorithm at some later time; the node which eventually detects termination should then inform the other nodes. Since competing nodes are eliminated very quickly, this also reduces the number of messages—after a short number of rounds with competing initiators, only one node remains which may repeatedly execute the algorithm. However, a drawback of this scheme is that it is no longer fully symmetric because the other processes are prohibited from initiating the termination test again. (Instead of that, however, they may ask the winner to initiate the test on behalf of them.) If this is a problem, the behavior of a node could be changed in a way that when it is visited by the (newer) query wave of another node it will not start an execution of the detection algorithm until it receives a special *wake-up signal* from the winning node. The winning node should initiate a timestamped wake-up wave if it does not detect global

termination (line 21). Each node that wakes up propagates the wake-up signal, echoes are not necessary for that purpose. The timestamps are used to absorb secondary wake-up signals and to reduce the traffic caused by outdated wake-up waves.

6. The Second Algorithm—Indulgent Process Priority. In order to avoid the problem of "mutual revenge" and to keep the scheme symmetric without synchronizing the nodes by a wake-up wave, we slightly modify the algorithm by assigning *fixed priorities* based on process numbers to the nodes. A parameter INIT of a query wave denotes the identity of its initiator and a process variable ENGAGER (initialized to 0) is used to hold the identity of the wave the node is currently engaged with. When a lower priority wave reaches a node engaged in a higher priority wave it immediately retreats (line 9) by generating an echo with the communication flag set to indicate that the test failed. If a higher priority query message reaches a node currently engaged with a lower priority wave (INIT > ENGAGER $\neq 0$), the acceptance of that message is deferred until the node is eventually disengaged (ENGAGER = 0, line 15). A node which is currently not engaged may start the algorithm at the label START (line 5) after setting N and PRED to 0 and setting ENGAGER to its own process identification number. As before, basic messages are counted by S and registered by the COM flag.

```
Upon receiving QUERY(INIT) from process p when ENGAGER = 0:
      /* disengaged node—primary query */
 1.   if |NEIGHBORS| = 1 then   /* terminal node */
 2.     send ECHO(S, false) to p;
 3.   else
 4.     ENGAGER ← INIT; PRED ← p; N ← 1;   /* initialize */
 5.     START: ACCU ← 0; COM ← false;
 6.     send QUERY(ENGAGER) to NEIGHBORS − {PRED};
        /* propagate */
 7.   fi;

Upon receiving QUERY(INIT) from process p when INIT = ENGAGER:
 8.   CHECK_COMPLETED;   /* secondary query */

Upon receiving QUERY(INIT) from process p when INIT < ENGAGER:
 9.   send ECHO(0, true) to p;   /* repell/retreat */

Upon receiving ECHO(SUM, FAILED):
10.     ACCU ← ACCU + SUM;   /* accumulate */
11.     COM ← COM or FAILED;
12.     CHECK_COMPLETED;

procedure CHECK_COMPLETED:
13.   N ← N + 1;   /* count control messages */
```

```
14.  if N = |NEIGHBORS| then   /* complete */
15.     ENGAGER ← 0;   /* disengage */
16.     if PRED ≠ 0 then send ECHO(ACCU + S, COM) to PRED;
17.     elseif ACCU + S = 0 and not COM then terminated;
18.     else failed;
19.     fi;
20.  fi;
end CHECK_COMPLETED;
```

The main difference between this algorithm and the previous version is that a higher priority wave does not extinguish a lower priority wave—the lower priority wave is repelled and the echoes are allowed to run till completion instead, thus informing the initiator and avoiding the danger of an endless waiting. A drawback of this indulgent wave propagation scheme is its increased delay, but the assignment of fixed priorities is a simple principle to avoid the starvation and "after you after you" blocking problems.

It should be noted that our priority-based wave propagation scheme (and also the variant described at the end of the previous section) is basically a *distributed election algorithm*: several nodes may asynchronously start the algorithm, eventually the wave initiated by the highest qualifying node succeeds in traversing the whole graph. The links traveled by the respective echo messages constitute a (rooted) *spanning tree.*

7. Variations, Optimizations, and Discussion. If an execution of the algorithm presented in Section 6 failed because the flag is set, it is not possible to decide whether the wave had a collision with a higher priority wave or whether some nodes received basic messages while they were engaged (i.e., the underlying computation was not terminated). To overcome this problem, two different flags could be used instead. In order to minimize the number of collisions, a process which was engaged with a defeated wave and later disengaged by its own retreating echo wave should not accept any lower priority messages or start a new wave while it is waiting for a higher priority wave (which will eventually arrive). Instead, it should repell all lower priority query messages as if it was already engaged with the higher priority wave. The realization of this scheme is straightforward: the echo wave carries with it the value of the highest priority wave encountered so far. The echo wave of the winning node will then reset that value.

Compared with the original version of our first algorithm where starvation is possible (Section 3), the second variant (Section 6) is "less symmetric" since not every node has the same fair chance of winning the election. The *notion of symmetry* in distributed programs has recently received some attention [1], [13] and it is generally agreed that symmetry is a very useful concept enabling general and robust solutions. But it has also been observed that symmetry is a potential source of deadlocks, since in a fully symmetric system nothing prevents all processes from doing the same thing at the same time. Solutions which use the

static ordering of process numbers as a means for breaking symmetry are no longer symmetric in the general sense that "all processes have the same rights and duties" [2]. Since fairness ("any process may be elected") is not an important issue here, we confine ourselves to the still very useful principle of *syntactic symmetry.*

In principle, the problem of collisions due to concurrent and independent activations can be overcome by *any* (inherently symmetric) *election algorithm* which is run prior to the start of the actual detection algorithm and which determines a unique node as the single initiator for termination detection. Elections in general asynchronous networks are known to be of $O(e+n \log n)$ message complexity [12] where e denotes the number of edges and n the number of nodes. In [14] an election algorithm is proposed which can be initiated at any subset of k nodes and requires at most $2e+3n \log_2 k+O(n)$ messages. Because our *built-in election scheme* uses up to $2ke$ messages, it seems only practicable for a small number of concurrent initiators. However, we can estimate that when using the optimizations described at the beginning of this section the *average* message complexity of one round with k concurrent initiators is bounded by $O(e \log k)$ which is favorable for sparse graphs where $e = O(n)$. The argument is as follows: any node is reached by (at most) k different waves $w_1, \ldots, w_k$ (w_i denotes the identity INIT of the wave generated by the ith initiator). The arrival sequence is a random permutation $w_{\sigma(1)}, \ldots, w_{\sigma(k)}$. At any node, wave $w_{\sigma(j)}$ is only propagated once if it is larger than all previous waves that reached the node, i.e., if $w_{\sigma(j)} = \max\{w_{\sigma(1)}, \ldots, w_{\sigma(j)}\}$. The probability for that is $1/j$; summing up over all $j = 1, \ldots, k$ results in $H_k \approx 0.58 + \log k$, the kth harmonic number. When propagating the wave, the node sends a query message to all of its (other) neighbors, their number is $2e/n - 1$ on the average. For all n nodes the average number of query messages is therefore bounded by about $2e \log k$. The number of echoes is bounded by the number of query messages.

One definitive advantage of our built-in election scheme is that it runs in combination with the termination detection algorithm. No separate passes for election, termination detection, and possibly opening of the next election round are necessary. However, if the number of concurrent initiators is expected to be high and competing nodes are not eliminated as described in Section 5, it might be favorable to determine a spanning tree once and use it subsequently for the election of a leader. Also we should be aware that the echo principle probably becomes impractical if the number of neighbors of a node is generally large or if every node can communicate directly with every other node. Since it is essential that control messages travel along every edge, the number of messages would then be of the order of $O(n^2)$ for each round. In that case a variant of the method operating on a predetermined spanning tree (or a Hamiltonian circle) and requiring two complete traversals can be used. The principles of this variant are described in [15] and [18].

A drawback of the termination detection method is its unbounded *message complexity.* The problem is to decide when to start a next trial if termination could not be established because of the counter being greater than 0 although the communication flag was not set. In that case some slow basic messages were

still in transit, but it is not known at which node to expect them and it cannot be guaranteed that they will have arrived before the next round. However, by equipping each node with incoming and outgoing message counters for each communication link it is possible to guarantee that during each detection round (except the last one) at least one basic message is received. This leads to an upper bound of $2e(m+1)$ control messages for one initiator where m denotes the number of basic messages. To achieve that, a control message carries with it the number of basic messages which have previously been sent over the communication link. The control message will only be accepted at the other end if at least that number of basic messages have been received. It should be noted, however, that it is extremely unlikely that only one basic message is in transit during each control round. On the other hand, the *best case* is independent of m; in fact, it is possible that termination is detected in *one single round* using $2e$ control messages. The average or typical case is difficult to estimate, but only few control messages should be generated if a process propagates the control messages only when it is passive (and, possibly, if its local state is consistent with a global termination state) and if the next round is only started when the initiator has reasons to suspect that global termination has occurred.

Since global knowledge such as the structure of the network or the total number of processes is not used, the algorithms are easily adaptable to *dynamic systems.* If new communication links (or links to new process nodes) are established, the identities of the new links or nodes are merely inserted into the NEIGHBORS set. If this happens while the process is engaged with a termination detection phase it has to send a query message along the new link. Of course, some care has to be taken when removing processes or links in order not to wait for messages from processes which no longer exist or to send messages to already dead processes.

When communication is *synchronous*, the number of messages in transit on a communication link is always 0 which simplifies the general principle of termination detection. The message counters can then be removed from our algorithms since whenever the accumulated counter is different from 0, the communication flag is set: the arguments of Section 4 show that if the flag is not set, only messages sent by a white process and received by a green process can disturb the balance of the counter; but since a white process never has a green neighboring process, *synchronous* communication cannot take place between a white and a green process. Therefore, it suffices to keep the flag.

Finally, the issue of *fault-tolerance* should be briefly addressed. We will see that apart from extreme cases (network partitioning) the algorithm can cope well with network failures. We may assume that transient network failures are handled by the underlying communication system using techniques such as checksums, acknowledgments, and a timeout and retransmit protocol. Because lost *basic messages* are considered to be in permanent transit by the termination detection protocol, we must also assume that the sender (or some other process) is eventually notified if a basic message could not be delivered—otherwise termination can never be determined. The loss of basic messages may also be determined by the underlying application using timeouts. Independent of the recovery action the

underlying application may then take, the message counter S must be corrected accordingly in some process (typically the sender or the receiver).

On the other hand, if a *control message* is lost the termination detection algorithm blocks, since each single control message is essential for the progress of the (single initiator) echo scheme. Again, this might be detected by timeouts or by notification by the underlying communication system. Because of the symmetry of the scheme, any process which becomes aware of such a problem may simply restart the detection algorithm, possibly after the communication system has closed (i.e., removed) the faulty communication link. In order not to intermingle the control messages, sequence numbers should be used to invalidate the messages of the previous control rounds. Therefore, as long as the network remains connected, a communication link may fail at any time; the algorithm will adapt itself to the new topology. However, if the network is partitioned or a process fails (i.e., becomes permanently unreachable), important information is lost and detection of termination (on the basis of the processes which are still reachable) is not possible by our scheme. Because the echo wave can easily count the number of reachable nodes, however, the initiator can detect these events, if it knows the total number of nodes.

8. Related Work. In recent years over 50 papers on the distributed termination problem have been published [18]. Some of the solutions of these papers are based on similar principles or have partially comparable properties. As already mentioned before, Kumar [15] and Lai [16] have also used the *principle of message counting* to detect termination of asynchronously communicating processes. However, their schemes are not symmetric. Lai makes use of timestamped basic messages; control messages are generated by a fixed initiator and travel along the edges of a predetermined tree. Concurrent activations are also not possible in [15] where fixed cycles are used for the control messages and generally two rounds are necessary, one for collecting the values, and another for testing the flag. More algorithms based on the message-counting principle are presented in [18].

Shavit and Francez [23] describe a generalized and symmetric version of the Dijkstra-Scholten principle [9] which views a distributed execution as a collection of diffusing computations which is tracked by a scheme reminiscent of the echo principle. The worst-case message complexity is $O(m+ne)$. However, their scheme uses *at least* m control messages, whereas our best case is independent of the number of basic messages and can be much smaller than m.

All other known solutions to the distributed termination problem require the FIFO property or are based on synchronous communication. In [25], Tan and van Leeuwen present symmetric solutions to the termination problem which are directly based on distributed election protocols and also make use of the principle of wave extinction. Although the general principle for symmetry breaking is similar to our scheme, the two approaches are rather different in detail; in particular, the termination-detection criteria are different (while our method is based on distributed message counting and consistency checking, their scheme

is based on the "black-white paradigm" as found in [10] and [27]). The underlying computation is assumed to be synchronous and control communication must observe the FIFO discipline. Other symmetric solutions based on synchronous communication are even more restrictive; Richier [21] assumes the existence of a Hamiltonian circuit for the propagation of control messages, Skyum and Eriksen [24] assume that an upper bound for the diameter of the communication graph is known to all processes.

For synchronous communication our principle is also reminiscent of the method described by Topor [27] and Francez and Rodeh [11]. However, these solutions are not symmetric. A fixed spanning tree is used and only the root process of that tree can initiate the algorithm.

Finally, it should be mentioned that some basic principles of our algorithms were already used in applications other than termination detection. The principle of *wave extinction* was used by Chang and Roberts in their election algorithm for rings of processes [7] and in the deadlock-detection algorithm by Chandy *et al.* [6]. Another distributed deadlock-detection algorithm [3] based on FIFO communication makes use of the *echo principle* where any node may initiate a wave tagged with the initiator identity and a sequence number. However, since the principle of wave extinction is not used, the message-load is not reduced in the case of contemporary initiations. Conflict resolution mechanisms for distributed systems employing *timestamp-based priority schemes* are a well-known principle [17], among others they are used in mutual-exclusion algorithms [20] and in concurrency control schemes [26].

9. Conclusions. We presented two variants of the general principle, both having a minor but specific drawback compared with the other. For the first variant the absence of starvation caused by repeated mutual extinction cannot be guaranteed if competing nodes are not eliminated or if wake-up synchronization signals are not used; timeouts should then be used to reduce the frequency of initiations. The second variant suffers from increased delay of the prevailing detection wave if several executions of the algorithm are concurrently active.

The general method has a number of interesting properties. Its main advantages are:

It is suited for asynchronous communication without FIFO property.

It is not based on a predefined communication structure (ring, tree, ...), additional communication channels solely for control communication are not required.

Global knowledge (e.g., the total number of processes) is not used.

It is easily adaptable to dynamic systems.

It copes well with network failures.

The scheme is symmetric—every process executes an identical algorithm and repeated initiation by any node is possible.

Mutual exclusion or synchronization prior to the initiation of the algorithm is not required.

It allows parallel and distributed evaluation of the termination condition.

The total number of necessarily sequential control message transfers of one run (i.e., the time complexity) is proportional (with a factor less or equal to 2) to the diameter of the communication graph.

The total number of control messages for one round initiated by a single node (no collisions and wave extinctions) is $2e$, the expected number of messages increases only logarithmically with the number of concurrent initiators.

Control messages are small, their length is not dependent on the total number of processes.

Local computations for the processing of control messages are short and performed essentially when the process is passive with respect to the underlying computation, causing only negligible overhead.

The scheme is simple and readily implementable, nearly without any influence on the underlying computation.

The purpose of this paper was not only to present a new and efficient method of termination detection based on message counting in combination with the echo technique of parallel graph traversal, but also to generalize the principle of wave extinction in order to allow concurrent initiations of echo algorithms without prior synchronization or mutual exclusion leading to symmetric solutions. Since the only requirements are bidirectional communication links, the general principle of our algorithms should also be applicable to other areas of interest in distributed computing such as distributed deadlock detection and global snapshot evaluation.

Acknowledgments. The author would like to thank Christian Beilken, Michel Raynal, Mike Spenke, Gerard Tel, and Dieter Zöbel for helpful comments on an earlier version of the paper. Several useful suggestions were also provided by the referees.

References

[1] L. Bougé, Symmetry and genericity for CSP distributed systems, Report 85-32, LITP, University of Paris 7, 1985.

[2] L. Bougé, On the existence of symmetric algorithms to find leaders in networks of communicating sequential processes, *Acta Inform.* (to appear).

[3] G. Bracha and S. Toueg, Distributed deadlock detection, *Distrib. Comput.*, **2** (1987), 127–138.

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

[5] K. M. Chandy and J. Misra, A paradigm for detecting quiescent properties in distributed computations, in *Logics and Models of Concurrent Systems* (K. R. Apt, ed.), Springer-Verlag, Berlin, 1985, pp. 325–341.

[6] K. M. Chandy, J. Misra, and L. M. Haas, Distributed deadlock detection, *ACM Trans. Comput. Systems*, **1** (1983), 144–156.

[7] E. Chang and R. Roberts, An improved algorithm for decentralized extrema-finding in circular configurations of processes, *Comm. ACM*, **22** (1979), 281–283.

[8] E. J. H. Chang, Echo algorithms: depth parallel operations on general graphs, *IEEE Trans. Software Engng.*, **8** (1982), 391–401.

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

[10] 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.

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

[12] E. Gafni, Improvements in the time complexity of two message-optimal election algorithms, *Proc. 4th ACM Symp. on Principles of Distributed Computing*, 1985, pp. 175–185.

[13] R. E. Johnson and F. B. Schneider, Symmetry and similarity in distributed systems, *Proc. 4th ACM Symp. on Principles of Distributed Computing*, 1985, pp. 13–22.

[14] E. Korach, S. Kutten, and S. Moran, A modular technique for the design of efficient distributed leader finding algorithms, Technical Report, Computer Science Department, Technion, Haifa, 1986.

[15] D. Kumar, A class of termination detection algorithms for distributed computations, in *Proc. 5th Conf. on Foundations of Software Technology and Theoretical Computer Science* (N. Maheshwari, ed.), Springer-Verlag, Berlin, LNCS 206, 1985, pp. 73–100.

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

[17] L. Lamport, Time, clocks and the ordering of events in a distributed system, *Comm. ACM*, **21** (1978), 558–565.

[18] F. Mattern, Algorithms for distributed termination detection, *Distrib. Comput.*, **2** (1987), 161–175.

[19] F. Mattern, Experience with a new distributed termination detection algorithm, in *Proc. 2nd Int. Workshop on Distributed Algorithms* (J. van Leeuwen, ed.), Springer-Verlag, Berlin, LNCS 312, 1988, pp. 127–143.

[20] G. Ricart and A. K. Agrawala, An optimal algorithm for mutual exclusion in computer networks, *Comm. ACM*, **24** (1981), 9–17.

[21] J.-L. Richier, Distributed termination in CSP, symmetric solutions with minimal storage, in *Proc. STACS 85* (K. Mehlhorn, ed.), Springer-Verlag, Berlin, LNCS 182, 1985, pp. 267–278.

[22] A. Segal, Distributed network protocols, *IEEE Trans. Inform. Theory*, **29** (1983), 23–35.

[23] N. Shavit and N. Francez, A new approach to detection of locally indicative stability, Report RC 11925, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1986.

[24] S. Skyum and O. Eriksen, Symmetric distributed termination, in *The Book of L* (G. Rozenberg and A. Salomaa, eds.), Springer-Verlag, Berlin, 1986, pp. 427–430.

[25] R. B. Tan and J. van Leeuwen, General symmetric distributed termination detection, Report RUU-CS-86-2, Department of Computer Science, University of Utrecht, 1986.

[26] R. H. Thomas, A majority consensus approach to concurrency control for multiple copy databases, *ACM Trans. Database Systems*, **4** (1979), 180–209.

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