Distributed Computing (1987) 2:161–175 

# Algorithms for distributed termination detection*

**Friedemann Mattern**

Department of Computer Science, SFB124, University of Kaiserslautern,
P.O. Box 3049, D-6750 Kaiserslautern, Federal Republic of Germany

**Abstract.** The termination problem for distributed computations is analyzed in the general context of asynchronous communication. In the underlying computational model it is assumed that messages take an arbitrary but finite time and do not necessarily obey the FIFO rule. Time diagrams are used as a graphic means of representing the overall communication scheme, giving a clear insight into the difficulties involved (e.g., lack of global state or time, inconsistent time cuts) and suggesting possible solutions.
Several efficient algorithms for the solution of the termination problem are presented. They are all based on the idea of message counting but have a number of different characteristics. The methods are discussed and compared with other known solutions.

**Key words:** Distributed termination – Termination detection – Asynchronous communication systems – Distributed programming – Decentralized control – Atomic model of computation – Global snapshots – Communication deadlock detection – Global quiescence – Diffusing computation

* This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the SFB124 research project "VLSI-Design and Parallelism"

Distributed Computing (1987) 2:161–175

DISTRIBUTED COMPUTING



# Algorithms for distributed termination detection*

**Friedemann Mattern**

Department of Computer Science, SFB124, University of Kaiserslautern, P.O. Box 3049, D-6750 Kaiserslautern, Federal Republic of Germany

Friedemann Mattern *received the Diploma in computer science from the University of Bonn, West Germany, in* 1983. *He is now a research scientist in the Department of Computer Science at the University of Kaiserslautern and is currently completing his Ph.D. His primary research interests include distributed algorithms, programming language design, and compiler construction. The author can be reached by electronic mail via mattern@incas.uucp or mattern%uklirb.uucp@Germany.csnet.*

**Abstract.** The termination problem for distributed computations is analyzed in the general context of asynchronous communication. In the underlying computational model it is assumed that messages take an arbitrary but finite time and do not necessarily obey the FIFO rule. Time diagrams are used as a graphic means of representing the overall communication scheme, giving a clear insight into the difficulties involved (e.g., lack of global state or time, inconsistent time cuts) and suggesting possible solutions.

Several efficient algorithms for the solution of the termination problem are presented. They are all based on the idea of message counting but have a number of different characteristics. The methods are discussed and compared with other known solutions.

**Key words:** Distributed termination – Termination detection – Asynchronous communication systems – Distributed programming – Decentralized control – Atomic model of computation – Global snapshots – Communication deadlock detection – Global quiescence – Diffusing computation

* This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the SFB124 research project "VLSI-Design and Parallelism"

## 1 Introduction

In distributed systems, where processes communicate solely by messages, the detection of termination of a distributed computation is non-trivial since no process has complete knowledge of the global state, and global time does not exist a priori. The *distributed termination problem* was brought into prominence by Francez (1980), and since then, numerous solutions with different characteristics have been proposed. Most of these solutions are based on CSP or some other model of distributed computation with a *synchronous* message passing scheme. In contrast to this we will discuss the distributed termination problem in the more general context of *asynchronous* communication, where transmission delays are arbitrary but finite and messages are not necessarily received in the same order in which they were sent.

A distributed computation is considered to be *globally terminated* if every process is (locally) terminated and no messages are in transit. "Locally terminated" can be understood to be a state in which a process has finished its computation and will not restart any action unless it receives a message. This abstract view of distributed termination is equivalent to a *global deadlock state* in the communication deadlock model discussed by Chandy et al. (1983) and Chandy and Misra (1985). It is merely a matter of interpretation as to whether a system is terminated or deadlocked; the more

abstract and general term *global quiescence* is now sometimes used to denote such situations (Chandy and Misra 1986; Shavit and Francez 1986). The *problem* is to design a scheme which will detect global termination or quiescence by means of additional *control messages*.

Termination, as we understand it, is a property of the *global state* of a distributed computation. The termination predicate is *persistent*, i.e., once such a state has been reached, it will "never" change. Our communication oriented termination model in which each process is blocked and every communication channel is empty, contrasts with a more general but also more application dependent termination model, in which a distributed computation is considered to be finished, if some arbitrary predicate on the global state has been reached.

Persistent predicates of the global state in distributed systems can be tested by *distributed snapshot algorithms*; the basic algorithm (Chandy and Lamport 1985) is easily adapted for asynchronous message passing schemes without FIFO characteristic. Such a snapshot algorithm may in principle be used to detect the communication oriented termination property (Bougé 1985b). However, because termination is a more specialized global predicate, our well tailored algorithms are simpler and more efficient than augmented snapshot algorithms.

## 2 The atomic model of distributed computation

A distributed system consists of a fixed set of processes which communicate exclusively by *messages*. All messages are received correctly after an arbitrary but finite delay, and communication is *asynchronous*, i.e., a process never waits for the receiver to be ready before sending a message. Therefore, the communication system is assumed to have unlimited buffer capacity (or, taking a more realistic view, to maintain a transparent flow control mechanism). It is *not* required that messages sent over the same communication channel obey the *FIFO rule*.

We first present the characteristics of a *transaction oriented model* of distributed computation similar to an earlier one used in other termination detection methods (e.g., Dijkstra et al. 1983; Topor 1984):

(1) At any given time, a process is either active or idle.
(2) Only idle processes may receive messages. (This simplifies presentation of the detection algorithms. It is not a serious restriction, since due to (5) a process may change from active to idle shortly before receiving a message. The condition can also be fulfilled by buffering incoming messages and considering this message buffer to logically belong to the communication system).
(3) On receipt of a message a process may change from idle to active.
(4) Only active processes may send messages. (Since we are not concerned with the initialization problem, we assume that all processes are initially idle and a message arrives from outside the system to start the computation; alternatively we may assume that initially a process may be either active or idle).
(5) A process may change from active to idle at any time.

We confine ourselves to systems in which every process will eventually become idle, although this property is generally undecidable. It can easily be verified that if one of the termination detection algorithms is applied to a system in which some processes remain in their active state forever, the algorithm itself will not terminate.

If a process is only active during a finite time interval $[t_1, t_2]$, the exact duration of its active phase is irrelevant. Because message delay is arbitrary, all messages sent during this interval could also be sent at the beginning of the phase, taking a little longer on their way to the destination process (Figs. 1, 2).

A convenient method of representing the dynamic overall communication scheme is by means of *time diagrams*: For each process a horizontal line is drawn parallel to an imaginary global time axis and the activity of the process is emphasized. In the case of the *atomic model* this is indicated

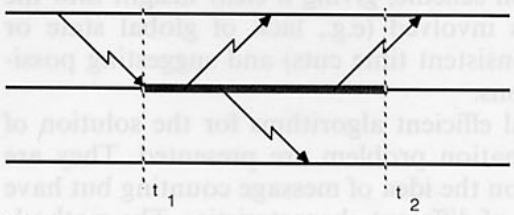

Fig. 1

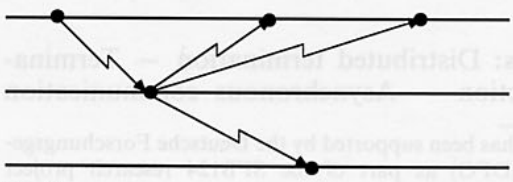
Fig. 2

by a dot. Messages are drawn as arrows, going from one activation spot to another further to the right. A *time cut* is regarded as a line crossing all process lines; ideally (when relying on "global time" in a sense that will become clearer later) this would be a straight vertical line. More formally, we can regard the time cut of a distributed computation in the atomic model as a *set of actions* characterized by the fact that whenever an action of a process belongs to that set, all previous actions of the same process also belong to the set.

By assuming that all local actions are performed in zero time, we get rid of the two process states "active" and "idle". In the *atomic model* of distributed computation, a process may at any time take any message from one of its incoming communication channels (provided one exists), immediately change its internal state and at the same instant send out any number of messages, possibly none at all (otherwise the computation will not terminate). To take a more pictorial view of the atomic model, messages can be thought of as flowing steadily but with various speeds towards their destination, eventually hitting a process. Then either nothing happens and the message is quietly consumed, or new "particles" are ejected, as if from an atomic reaction. For convenience, we assume that all atomic actions (including those of the superimposed detection algorithm) are totally globally ordered, i.e., no two actions occur at the same time instant.

Provided that every local computation initiated by the receipt of a message terminates, the transaction oriented model is equivalent to the atomic model of distributed computation. However, the atomic model, which is similar to the *actor* message-passing model (Clinger 1981), is especially appropriate when reasoning about distributed systems, because there is no concurrent activity of processes. This is in contrast with the synchronous CSP-model, in which message passing is instantaneous but processes are concurrently active. We consider the atomic model to give us a better insight into the problem of distributed termination and to assist us in finding solutions for it.

## 3 The problem

In the atomic model a distributed system is *terminated* at time $t$, if at this instant all communication channels are empty, i.e., if there is no message in the system which has been sent but not yet received.

In Fig. 3 the system is not terminated at $t_2$ because two messages on their way to $P_4$ have not yet

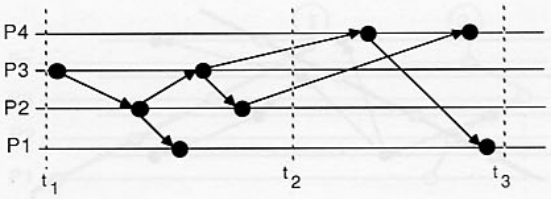

Fig. 3

been received. The system is terminated at $t_3$ since no message crosses the vertical line representing the time cut. The system is also terminated at $t_1$, but this is an initialization problem we will not discuss further. Even if the time cut is represented by a zigzag (Fig. 6), the system is terminated at the time fixed by the rightmost point ($t_4$ in Fig. 6) if no messages are crossing it. Obviously there can be no process activity to the right of such a time cut.

Let us assume that one dedicated process $P_i$, the initiator, wants to know if the computation has terminated. Because it has no global view of the system, two *problems* arise:

(1) It can not ask the communication medium or the channels whether there are any messages still to be received, since information can only be obtained from processes.

(2) If $P_i$ sends *control messages* to all processes then they are all received at different times. (These additional control messages are distinct from the basic messages of the underlying application).

Let us assume without loss of generality that processes $P_1, \ldots, P_n$ are ordered in sequence of the arrival of the control message and that $P_1$ is the initiator sending control messages directly or indirectly to all other processes.

To find out whether any messages have not yet been received, the most obvious solution is to let every process *count* the number of sent and received basic messages. We denote the total number of basic messages $P_i$ has sent at (global) time instant $t$ by $s_i(t)$, and the number of messages received by $r_i(t)$. The values of the two local counters are communicated to the initiator upon demand. Having directly or indirectly received these values from all processes the initiator can accumulate the counters. In Fig. 4 the time instants at which the

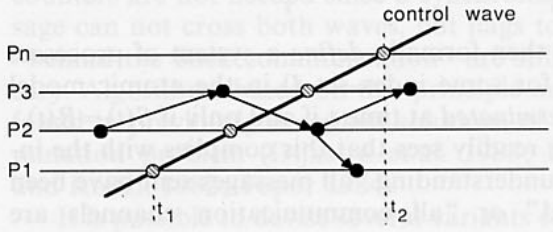

Fig. 4

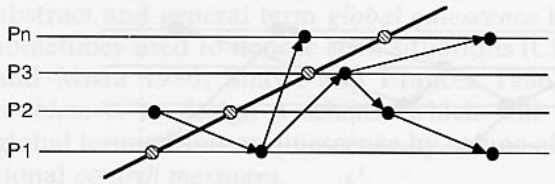


Fig. 5

processes receive the control messages and communicate the values of their counters to the initiator are symbolized by striped dots. These are connected by a line (representing a "*control wave*") which induces a time cut. In the example shown in Fig. 4 the accumulated values indicate that two messages were sent and one message was received. The initiator obtains the accumulated result after $t_2$ and notices that the system has not been terminated. This was because a message crossed the control wave, possibly initiating some communication to its right ("behind the back communication").

Unfortunately because of the time delay of the control wave, this simple algorithm is *not correct*. Figure 5 shows that the counters can become corrupted by messages "from the future", crossing from the right side of the control wave to its left.

The accumulated result indicates that one message was sent and one was received – although the computation has not yet actually terminated. The cause of this misleading result lies in the inconsistent time cut. A time cut is considered to be *inconsistent*, if when the diagonal line representing it is made vertical, by compressing or expanding the local time scales (without changing the relative order of the local events of a process), a message crossing the control wave is found to travel backwards in time.

Using the set characterization of a time cut outlined in Chapter 2 we can *define* a time cut to be *inconsistent* if a message from an action outside the time cut set activates an action belonging to the set.

In order to analyze the problem in more detail, we denote the total number of messages sent and received by all processes at time $t$ by

$S(t) := \sum_i s_i(t)$ and $R(t) := \sum_i r_i(t)$ respectively.

We then formally *define* a system of processes $P_i$ ($i \in I$ for some index-set $I$) in the atomic model to be *terminated* at time $t$ if and only if $S(t) = R(t)$.

One readily sees that this complies with the intuitive understanding "all messages sent have been received" or "all communication channels are empty".

Since the control messages of the "naive counting method" are received by the processes $P_i$ at *different* time instants $t_i$, the initiator wrongly compares

$S^* := \sum_i s_i(t_i)$ to $R^* := \sum_i r_i(t_i)$

instead of comparing $S(t)$ to $R(t)$ for some fixed time instant $t$.

*Global time* in a distributed system is merely a myth, since a distributed system allows each observer his or her own global time. The same computation can be depicted by different but *equivalent* time diagrams. These differ in the absolute global time instants assigned to the atomic actions but preserve the relative order of the local events of a process as well as the activation ordering. The latter represents the communication between actions; an action is activated by the receipt of a message which was sent by another *previously* active action – messages are always received *after* they were sent. The invariance of the local event ordering and the activation ordering amongst all possible "observations" (i.e., time diagrams) of a computation is fundamental for preserving causality (Lamport 1978; Clinger 1981).

In situations for which an equivalent time diagram exists, where all processes receive the control messages at the same time instant $t_0$, we obviously have a *consistent* time cut. In that case $S^* = S(t_0)$ and $R^* = R(t_0)$ and consequently $S^* = R^*$ implies $S(t_0) = R(t_0)$, i.e. termination at $t_0$. It should be clear now, that the naive counting method does work for consistent time cuts!

Various methods can be applied to correct the situation for general time cuts, some ideas are:

(1) *Detect* inconsistent time cuts and possibly restart the algorithm at a later time, i.e. detect the arrival of a message from the future (Chapter 6.1).

(2) *Avoid* inconsistent time cuts by designing a scheme which provides only consistent time cuts and prohibits messages from crossing the wave from right to left (Beilken et al. 1985).

(3) Do not rely on simple accumulated counters; treat the basic messages more *individually* in order not to relate the sending of one message to the receipt of another (Chapters 6.2 and 7).

(4) Impose *tighter control* on the basic messages (e.g., freeze basic communication for a certain time, acknowledge every single message directly, impose a FIFO-ordering ...). We will not discuss algorithms that rely on such methods, because they give rise to inherently inefficient solutions or seriously restrict our general model of distributed computation.

## 4 A solution – the four counter method

A very simple solution, which can be improved upon in various ways, consists of counting *twice* using the previously discussed (wrong) naive counting method and comparing the results. After the initiator has received the response from the last process and accumulated the values of the counters $R^*$ and $S^*$, it starts a second control wave, resulting in $R'^*$ and $S'^*$. We claim that the system is terminated, if the values of the four counters are equal, i.e. $R^* = S^* = R'^* = S'^*$.

In fact, we prove a slightly stronger result: If $R^* = S'^*$, then the system was terminated at the end of the first wave ($t_2$ in Fig. 6). A few very simple lemmata prepare the *proof* of this conjecture.

Let $t_2$ denote the time instant at which the first wave is finished, and $t_3 \geq t_2$ the starting time of the second wave (see Fig. 6).

(1) The local message counters are monotonic, $t \leq t'$ implies $s_i(t) \leq s_i(t')$ and $r_i(t) \leq r_i(t')$.

*Proof.* Follows from the definition.

(2) The total number of messages sent, or received, is monotonic, $t \leq t'$ implies $S(t) \leq S(t')$ and $R(t) \leq R(t')$.

*Proof.* Follows from the definition and (1).

(3) $R^* \leq R(t_2)$.

*Proof.* Follows from (1) and the fact that the values $r_i$ are collected before ($\leq$) $t_2$.

(4) $S'^* \geq S(t_3)$.

*Proof.* Follows from (1) and the fact that all values $s_i$ are collected after ($\geq$) $t_3$.

(5) For all $t$: $R(t) \leq S(t)$.

*Proof.* The nonnegative deficit $D(t) := S(t) - R(t)$ is the number of messages in transit; $D(t) \geq 0$ is an invariant (induction on the number of actions).

The proof of the conjecture now follows directly:

$$R^* = S'^* \Rightarrow R(t_2) \geq S(t_3) \quad (3,4)$$
$$\Rightarrow R(t_2) \geq S(t_2) \quad (2)$$
$$\Rightarrow R(t_2) = S(t_2) \quad (5)$$

i.e., the system is terminated at $t_2$.

If the system is terminated before the start of the first wave, it is trivial that all messages arrived and hence the values of the accumulated counters are identical. Therefore termination is detected by the algorithm after its occurrence within two "rounds".

It should be noted that the second wave of an unsuccessful termination test can be used as the first wave of the next trial. However, the problem with this method is to decide when to start the next wave after an unsuccessful test, since there is a danger of an unbounded control loop being set up. This possibility is avoided in the solutions presented in Chapters 6.2 and 7.

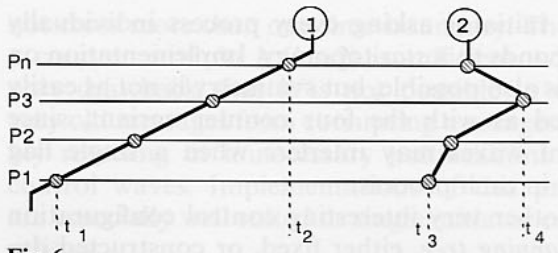

Fig. 6

One rather nice property of the method when implemented on a *ring* is its *symmetry*. In this case any process can start termination detection by sending out a ciruclating control message which collects the counters; synchronization or mutual exclusion is not necessary, and several control waves can be concurrently active. Furthermore, no process needs to know the total number of processes.

## 5 The sceptic algorithm and its variants

It is observed, that the values of the counters obtained by the first wave of the four counter method can become corrupted if there is activity at the right of the wave between $t_1$ and $t_2$ (the time delay of the wave, see Fig. 6). To detect such activity, it is also possible to use *flags* initialized by the first wave, and set by the processes when they receive (or alternatively when they send) messages. A second wave only needs to check if any of the flags have been set, in which case a possible corruption is assumed. Several variants of this "sceptic algorithm" can be devised (e.g., counting and checking the flags in a combined wave (Kumar 1985)), but a general drawback is that in general at least two waves are necessary to detect termination. Since the sceptic algorithm is merely a simple variant of the four counter method, a proof is not given.

In *synchronous* message passing models, counters are not needed since a synchronous message can not cross both waves, but flags to detect "behind the back communication" are still necessary. Algorithms based on this principle are to be found in previously described solutions of the termination problem (Dijkstra et al. 1983; Francez and Rodeh 1982; Topor 1984).

It is possible to devise several variants concerning the *logical control topology*:

An initiator asking every process individually corresponds to a *star* topology. Implementation on a *ring* is also possible, but symmetry is not as easily achieved as with the four counter variant, since different waves may interfere when a single flag is used in each process.

Another very interesting control configuration is a *spanning tree*, either fixed, or constructed dynamically (Lavallee and Roucairol 1986) or implicitly by an *echo algorithm* (Chang 1982). Echo algorithms used as a parallel graph traversal method induce two phases on a node: The "down" phase is characterized by the receipt of a first control message (which is propagated to all other neighbors), the "up" phase by the receipt of the last of the echoes from its neighboring nodes. (An echo travelling in the opposite direction to a control message is generated whenever the node is either a leaf and gets a control message, is still engaged with a previously received control message or when the last echo from its neighbors arrives). These two phases can be used as the two necessary waves of the sceptic termination detection method (Mattern 1986). The echo principle permits a decentralized implementation of the sceptic algorithm, allowing concurrency without as serious a bottleneck as with the star topology.

Static trees are used in termination detection algorithms by Topor, Francez and Rodeh for models based on synchronous communication (Topor 1984; Francez 1980; Francez and Rodeh 1982). The more interesting dynamic spanning trees are used in the diffusing computation scheme (Dijkstra and Scholten 1980; Misra and Chandy 1982; Shavit and Francez 1986) and also in the deadlock detection method by Chandy et al. (1983). A similar solution is presented in Chapter 7.

## 6 Single wave detection algorithms

Even if the system has already terminated when a sceptic algorithm is started, in general at least two waves are necessary. (If $S^*=0$ in the first wave, then the system was terminated initially; and, of course, a second confirmation wave is not necessary). At the expense of increasing the amount of control information or augmenting every message with a time stamp, termination can be detected in *one single wave* after its occurrence.

### 6.1 The time algorithm

In the time algorithm each process is equipped with a *local clock* represented by a counter initialized to 0.

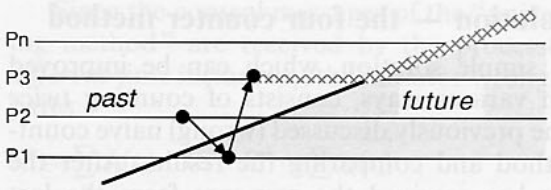


Fig. 7

A control wave started by the initiator at time $i$, accumulates the values of the counters and "synchronizes" the local clocks by setting them to $i+1$. In this way the control wave separates "past" from "future" (Fig. 7). A process is made aware of the fact that it has received a message from the future, i.e., one that crossed the wave from right to left and corrupted the counters, due to the *time stamp* of the message being greater than its own local time. After such a message has been received, the current control wave is nullified on arrival at the process (Fig. 7).

Since all inconsistent time cuts are detected, the fact that the method is correct can be deduced from the considerations at the end of Chapter 3. The realization of a single initiator variant is straightforward, here we give a slightly more complicated *symmetric* implementation on a ring of $n$ processes ($n>1$) where any process may start the termination test independently of all other processes.

Every process $P_j$ ($1 \le j \le n$) has a local message counter COUNT (initialized to 0) holding the value $s_j - r_j$, a local discrete CLOCK (initialized to 0) and a variable TMAX (also initialized to 0) holding the latest send-time of all messages received by $P_j$. $P_j$ behaves as follows:

(a) When sending a basic message to $P_i$:
1. COUNT ⟵ COUNT + 1;
2. **send**⟨CLOCK, ...⟩ **to** $P_i$;
/* time-stamped basic message */

(b) When receiving a basic message ⟨TSTAMP, ...⟩:
3. COUNT ⟵ COUNT − 1;
4. TMAX ⟵ max(TSTAMP, TMAX);
5. /* process the message */

(c) When receiving a control message ⟨TIME, ACCU, INVALID, INIT⟩:
6. CLOCK ⟵ max(TIME, CLOCK); /* synchronize */
7. **if** INIT = $j$ /* complete round? */
8. **then if** ACCU = 0 **and not** INVALID
**then** "*terminated*"
9. **else** "*try again*?"
10. **end_if**;

11. **else send**⟨TIME, ACCU + COUNT, INVALID **or** TMAX ≥ TIME, INIT⟩ **to** $P_{(j \bmod n)+1}$;
12. **end_if**;

(d) When starting a control round:

13. CLOCK ⟵ CLOCK + 1;
14. **send**⟨CLOCK, COUNT, **false**, $j$⟩ **to** $P_{(j \bmod n)+1}$;

A control message consists of four components. The first indicates the (local) time at which the control round was started, the second is the accumulator for the message counters, the third is a flag which is set when some process received a basic message from the future (TMAX ≥ TIME) and the last component is the identification of the initiating process. The first component of a basic message is always the time stamp.

It is easy to see that, for each single control wave, any basic message crossing the wave from the right side of its induced time cut to its left side is detected. It is also obvious that different control waves do not interfere, they merely advance the local clocks even further. Once the system is terminated, the values of the TMAX variables remain fixed and since for every process $P_j$, $\text{TMAX}_j \leq \max \text{CLOCK}_i$ $(1 \leq i \leq n)$, the process with the maximum clock value can detect global termination within one round. Other processes may need more rounds, but because of the clock synchronization in line 6, the different clock values should be rather close together. Optimizations concerning this point are possible (e.g., allowing only one active control wave per process or collecting the maximum clock value while visiting the processes).

Several variations of the time stamp principle are possible, in some of which it is endeavoured to maintain a *global virtual time* by synchronizing the local clocks with every basic message. This principle was inspired by Lamport (1978). Rana, Apt and Richier make use of it in their termination detection algorithms which rely on the synchronous CSP-model (Rana 1983; Apt and Richier 1985). Reminiscent of this principle are methods for obtaining consistent snapshots (Chandy and Lamport 1985). Lai proposes a snapshot-based algorithm for a general asynchronous and dynamic model using time-stamped messages (Lai 1985, 1986) which is similar to our principle.

The values held in the counters serving as clocks can be kept bounded, since if a new control wave is only started after the previous one has finished, messages travelling backwards in time can not cross more than one time-boundary. Therefore a counter modulo $k$ $(k \geq 2)$ is sufficient, with the only drawback that it is not possible to discern very old messages from corrupting messages, possibly resulting in unnecessary nullifications of the control waves. Implementations of this principle are especially well suited to single initiator configurations.

The *bounded clock-counter* variant of the time algorithm for $k$ time-zones can be realized on a star configuration (with $P_0$ in the center as the initiator) as follows. Each process $P_j$ $(1 \leq j \leq n)$ has a message counter COUNT initialized to 0, a boolean flag TIME_WARP initialized to false (set to true whenever a basic message from a future time-zone arrives) and a variable TIME_ZONE with possible values $0 \ldots k-1$, initialized to 0. Any process $P_j$ $(j \neq 0)$ behaves as follows:

(a) When sending a basic message to $P_i$:

1. COUNT ⟵ COUNT + 1;
2. **send** ⟨TIME_ZONE, ...⟩ **to** $P_i$;

(b) When receiving a basic message ⟨TSTAMP, ...⟩:

3. COUNT ⟵ COUNT − 1;
4. TIME_WARP ⟵ TIME_WARP **or** TSTAMP = (TIME_ZONE + 1) **mod** $k$;
5. /* process the message */

(c) When receiving a control signal from $P_0$:

6. **send** ⟨COUNT, TIME_WARP⟩ **to** $P_0$;
7. TIME_WARP ⟵ **false**;
8. TIME_ZONE ⟵ (TIME_ZONE + 1) **mod** $k$;

The initiator $P_0$ sends an otherwise empty signal to each $P_j$, waits for the replies and accumulates the reported message counters and TIME_WARP flags. It announces global termination if the accumulated counter equals 0 and all flags are false, otherwise it begins another query.

Note, that although the clock counters can be kept bounded, the same is not true for the message counters since our model allows an arbitrary number of basic messages to be in transit at any time.

## 6.2 *Vector counters*

A drawback of the time algorithm is the augmentation of every basic message with a time stamp. Its purpose is to detect messages from the future which corrupt the counters. Another method of termination detection consists of counting the messages in such a way, that it is not possible to mislead the accumulated counters. We first describe an im-

plementation of this method on a ring of $n$ processes and then prove that this idea is correct.

Every process $P_j$ $(1 \leq j \leq n)$ has a COUNT vector of length $n$, where COUNT$[i]$ $(1 \leq i \leq n)$ denotes the $i$-th component of the vector. A circulating control message also consists of a vector of length $n$. For each process $P_j$, the local variable COUNT$[i]$ $(i \neq j)$ holds the number of basic messages which have been *sent* to process $P_i$ since the last visit of the control message. Likewise, the *negative value* of COUNT$[j]$ indicates how many messages have been received from any other process. At any (global) time instant, the sum of the $k$-th components of all $n$ COUNT vectors including the circulating control vector equals the number of messages currently on their way to $P_k$ for some fixed $k$ $(1 \leq k \leq n)$. It is easily verified that this property is maintained *invariant* by the implementation shown here. (Remember that in the atomic model all local actions are assumed to be performed in zero time.)

For simplicity in exposition of this algorithm, we assume that no process communicates with itself and that $P_{n+1}$ is identical to $P_1$. An arithmetic operation on a vector is defined by operating on each of its components; and 0* denotes the null-vector.

Process $P_j$ behaves as follows:

{COUNT is initialized to 0*}

(a) When sending a basic message to $P_i$ $(i \neq j)$:

1. COUNT$[i] \leftarrow$ COUNT$[i]+1$;

(b) The following instructions should be executed at the end of all local actions triggered by the receipt of a basic message:

2. COUNT$[j] \leftarrow$ COUNT$[j]-1$;
3. **if** COUNT$[j]=0$ **then**
4. **if** COUNT $=0^*$ **then** "*system terminated*"
5. **else send** accumulate ⟨COUNT⟩ **to** $P_{j+1}$;
6. COUNT $\leftarrow 0^*$;
7. **end_if**;
8. **end_if**;

(c) When receiving a control message 'accumulate ⟨ACCU⟩':

9. COUNT $\leftarrow$ COUNT + ACCU;
10. **if** COUNT$[j] \leq 0$ **then**
11. **if** COUNT $=0^*$ **then** "*system terminated*"
12. **else send** accumulate ⟨COUNT⟩ **to** $P_{j+1}$;
13. COUNT $\leftarrow 0^*$;
14. **end_if**;
15. **end_if**;

An initiator $P_i$ *starts* the algorithm by sending the control message 'accumulate ⟨0*⟩' to $P_{i+1}$.

Some sort of mechanism not actually shown in the realization of the algorithm must be employed to ensure that *every process is visited at least once* by a control message, i.e., that the circulating control vector performs at least one complete round after the start of the algorithm.

Every process counts the number of outgoing messages individually by incrementing the counter indexed by the receiver's process number (line 1); the counter indexed by its own number is decremented on receipt of a message (line 2). When receiving the circulating control message, it accumulates the values (line 9). A check is then made (line 10) to determine whether any basic messages known to the control message have still not arrived. If this is the case (COUNT$[j]>0$), the control message is removed from the ring and regenerated at a later time (line 5) when all expected messages have been received. For this purpose, every time a basic message is received, a test is made to check whether COUNT$[j]$ is equal to 0 (line 3).

Note that lines 4–15 are only executed when the control vector is at $P_j$. In order to understand the algorithm it is helpful to realize that there is at most one process $P_j$ with COUNT$[j]>0$ (an easily verified property since no process communicates with itself), and that if this is the case, the control vector "parks" at the process (guard at line 10: line 12 is not executed and the control vector remains at $P_j$).

An interesting characteristic of this method is that the control wave *waits* for basic messages which take a long time. Another property of the algorithm discerning it from previously described methods, is that, once the control message has been started and has made one full round, it remains active until the distributed computation finishes and termination is reported. Termination can be detected by any process. The algorithm (at least the version presented here) must not be started more than once.

If it is not required that the control message waits at nodes for outstanding basic messages, the algorithm can be simplified considerably by removing lines 3–8 as well as lines 10 and 15. Other variants of the algorithm which allow processes to send messages to themselves, or there to be several concurrently active control waves, started by different

processes, as well as implementations using other logical configurations are readily devised.

The number of control messages of the algorithm is bounded by $n(m+1)$, where $m$ denotes the number of basic messages, because at least one basic message is received in every round of the control message, excluding the first round. The fact that after $k$ rounds at least $k-1$ basic messages have been received is easily *proved* by induction on the number of control cycles $k$: The base case for $k=1$ is trivial. Inductively, assume that after $k$ (where $k>0$) rounds have been performed and the control message is again at the initiator, at least $k-1$ basic messages have been received but termination has not yet been detected. Then there exists a basic message whose sending, but not receipt, was registered during the $k$-th round, i.e. a message that crossed the $k$-th control wave of the time diagram. As a feature of the algorithm, the control wave belonging to the next round waits at the receiver process for the expected message, if it has still not arrived by this time. This shows that at least one more message will be received during the $(k+1)$-th control cycle.

In order to prove the central idea of the algorithm, namely that the system is actually terminated at the instant when the accumulated vector becomes the null-vector, we examine the last cycle of the control message in the time diagram depicted in Fig. 8. (Remember that at least one round is completed.)

Assume that the accumulated vector at some process $P_j$ is the null-vector (i.e. the number of registered messages sent by each process equals the number of registered messages it received) but that there is activity on the right of the control wave. The earliest process ($P_i$ in Fig. 8) that becomes active after the control wave at time $t_2$ can become so only due to an activating message which crosses the wave. This means that the sending of the message is registered, but not its receipt. For a corruption of the counter COUNT[$i$] to take place, there must be a compensating message that crosses the wave from right to left whose arrival at $P_i$ is detected, but whose sending goes unnoticed. Such a message must be received before $t_1$ (the time instant at which the wave passes $P_i$) and consequently (because messages do not travel backwards in time) must also be sent out before $t_1$. However, every message sent before $t_1$ is registered by the control wave; no process activity exists at the right of the wave before $t_1$, because the first activation takes place at $t_2 > t_1$. We conclude that a message crossing the control wave affects the counters in such a way that no corruption is possible. Hence there is no possibility of detecting false termination.

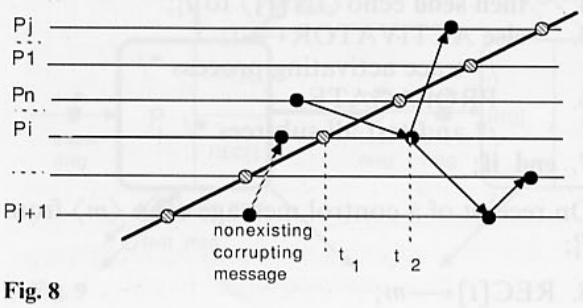


**Fig. 8**

## 7 Channel counting

In the vector counter method each message is counted twice; by its sender and by its receiver. The sender has individual counters indexed by the recipient of the messages, whereas the receiver does not differentiate between the senders.

The channel counting method can be regarded as a refinement of the vector counter method; the receiver takes note of the sender and keeps track of the number of messages received by each node, using the appropriate counter. We make use of the idea on a dynamically constructed *tree*. The principle is similar to a recently presented method (Kumar 1985) which is based on counting messages on every communication channel and a marker which traverses the network on a *cycle*.

Each process $P_j$ has $n$ counters, $C^+_{j1}, \ldots, C^+_{jn}$, for outgoing messages and $n$ counters, $C^-_{1j}, \ldots, C^-_{nj}$, for incoming messages. $C^-_{ij}$ is incremented on receipt of a message from process $P_i$, and $C^+_{jk}$ is incremented when sending a message to $P_k$. Upon demand, each process informs the initiator of the values of the counters. The initiator reports termination if $C^-_{ij} = C^+_{ij}$ for all $i, j$. It should be noted that the (virtual) communication channels from $P_i$ to $P_j$ are not necessarily empty if $C^-_{ij}$ equals $C^+_{ij}$ for *some* $i, j$; although it is claimed that this is the case if the condition holds for *all* $i, j$.

The proof that no false termination can be reported is left out, since it is very similar to the one for the vector counter method. It basically consists of showing that there is no compensating "corrupting" message for the message that activated the first process to the right of the last control wave (see also Kumar 1985).

The information available at the initiator for channels from $P_i$ to $P_j$ for which $C^-_{ij} \neq C^+_{ij}$ allows the dedicated revisiting of certain nodes. However, the channel counting method probably has no advantages over the vector counter method for ring and star topologies. Basic messages have to be

aware of their sender and the amount of control information is increased, since there are two counters for each edge of a fixed communication graph, or in the general case, $n^2$ counters.

The method actually becomes practicable if it is combined with the *echo algorithm* (Chapter 5) where test messages flow "down" on every edge of the graph and echoes proceed in the opposite direction. The value of the counter $C_{ij}^-$ is transmitted upwards from process $P_j$ to $P_i$ by an echo; whereas for reasons that will become clear later, a test message sent by $P_i$ to $P_j$ carries the value of $C_{ij}^+$ with it. (In the standard application of the echo algorithm all values of $C_{ij}^-$ and $C_{ij}^+$ would be transmitted upwards by echoes until all values are accumulated at the initiator). A process receiving a test message from another process (the "activator"), propagates it in parallel to any other processes to which it has sent basic messages whose receipts have not yet been confirmed. If it has already done this, or if all basic messages sent out have been confirmed, an echo is immediately sent to the activator. There are no special messages of acknowledgement. A process $P_i$ receiving the value of $C_{ij}^-$ by an echo, knows that all messages it sent to $P_j$ have arrived if the value of $C_{ij}^-$ equals the value of its own counter $C_{ij}^+$. An echo is only propagated towards the activator if an echo has been received from each active subtree and all channels in the subtrees are empty.

As in the similar diffusing computation scheme (Dijkstra and Scholten 1980), it is assumed that a special process, the initiator, starts the computation by sending messages to some other nodes in a single spontaneous event. It is also the initiator which later starts the distributed termination detection algorithm.

In the realization of our algorithm, shown further on in this chapter, each process $P_j$ has 3 arrays of counters:

OUT[$i$] counts the number of basic messages sent to $P_i$

IN[$i$] counts the number of basic messages received from $P_i$

REC[$i$] records the number of its messages $P_j$ is aware have been received by $P_i$.

OUT[$i$] corresponds to $C_{ji}^+$, IN[$i$] to $C_{ij}^-$. A variable ACTIVATOR is used to hold the index-number of the activating process (line 5) and the counter DEGREE is used to indicate how many echoes are still missing. It is incremented when sending a test message (line 20) and decremented on receipt of an echo (line 9). If DEGREE > 0, then the node is "engaged" and further activations by test messages are immediately reacted to with echo messages (line 4).

An echo is also immediately generated if OUT = REC (line 3), i.e., if the process sent no messages at all or if all messages sent out have been acknowledged. Lines 10–15 guarantee that an echo is only generated if the arrival of all basic messages has been confirmed and all computations in the subtrees finished. This is achieved by sending off further test messages after the last echo has arrived (lines 10–12). They revisit any of the subtree root processes which have not yet acknowledged all basic messages sent to them. The procedure PROPAGATE increases the value of the variable DEGREE if any processes are visited, thus prohibiting the generation of an echo (lines 13–15).

In order to minimize the number of control messages, test messages should not overtake basic messages. To achieve this, test messages carry with them a count of the number of basic messages sent over the communication channel (line 19). If a test message overtakes some basic messages (and it itself is not overtaken by basic messages), its count must be greater than the value of the IN-counter of the receiver process. In this case the test message is held back by some mechanism, and delivered at a later instant, when all basic messages have been received (guard $m \leq$ IN[$i$] in point $c$).

Process $P_j$ behaves as follows:

{OUT, IN, REC, DEGREE are initialized to 0* resp. 0}

(a) When sending a basic message to $P_i$:

1. OUT[$i$] ⟵ OUT[$i$] + 1;

(b) When receiving a basic message from $P_i$:

2. IN[$i$] ⟵ IN[$i$] + 1;

(c) On receipt of a control message test ⟨$m$⟩ from $P_i$ where $m \leq$ IN[$i$]:

3. **if** DEGREE > 0 **or** OUT = REC
   /* already engaged or */
   /* subtree is quiet */
4. **then send** echo⟨IN[$i$]⟩ to $P_i$;
5. **else** ACTIVATOR ⟵ $i$;
   /* trace activating process */
6. PROPAGATE;
   /* and test all subtrees */
7. **end_if**;

(d) On receipt of a control message echo ⟨$m$⟩ from $P_i$:

8. REC[$i$] ⟵ $m$;

```
9.  DEGREE←—DEGREE−1;
        /* decrease missing echoes counter */
10. if DEGREE=0 then
        /* last echo checks whether all */
        /* subtrees are quiet */
11.     PROPAGATE;
12. end_if;
13. if DEGREE=0 then
        /* all echoes arrived, everything quiet */
14.     send echo ⟨IN[ACTIVATOR]⟩
        to P_ACTIVATOR;
15. end_if;
```

(e) The procedure PROPAGATE called in lines 6 and 11 is defined as follows:

```
16. procedure PROPAGATE:
17.     loop for K=1 to n do
18.         if OUT[K]≠REC[K] then
                /* confirmations missing */
19.             send test ⟨OUT[K]⟩ to P_K;
                /* check subtree */
20.             DEGREE←—DEGREE+1;
                /* more echoes outstanding */
21.         end_if;
22.     end_loop;
23. end_procedure;
```

A snapshot of an execution of the algorithm is shown in Fig. 9.

The initiator starts the termination test only once, in the same way as if it had received a test⟨0⟩-message from some imaginary process $P_0$. Instead of eventually sending an echo to $P_0$, it reports termination. Test messages only travel along channels which were used by basic messages, nodes that did not participate in the distributed computation are not visited by test messages. For each test message, an echo is eventually sent in the opposite direction.

There must be at least one test message travelling along every channel previously used by basic messages. Therefore, if $e$ denotes the number of such channels, at least $2e$ control messages are generated. It is easy to see that if the termination test is started after the global termination condition has been satisfied, no channel is used by a test message more than once. Therefore, exactly $2e$ control messages are required in this case, some of them being concurrently active.

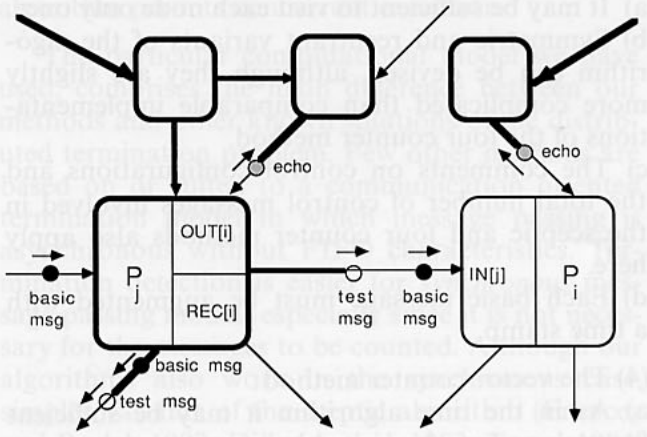


Fig. 9

At least one basic message must have been sent between the sending of two test messages along the same channel. This results in an upper bound of $2m$ control messages, where $m$ denotes the number of basic messages. This worst case should rarely occur, particularly if the termination test is started well after the computation proper. This contrasts the channel counting algorithm with the method proposed by Dijkstra and Scholten (1980) and Shavit and Francez (1986). In their method, test messages do not exist and each basic message is acknowledged by an echo; therefore the number of control messages is exactly $m$. The exact number of control messages involved in the channel counting method is difficult to estimate, but in reasonable situations should be much smaller than $m$.

Counting the sent and received messages individually per communication channel is also suggested by Chandy and Misra, but in their method the counters $C_{ij}^-$ and $C_{ij}^+$ are both located at process $P_i$ and each single basic message is acknowledged by an echo (Chandy and Misra 1985). The overall scheme of our channel counting method is also reminiscent of their distributed deadlock detection method (Chandy et al. 1983).

## 8 Discussion

Unlike most other works on distributed termination, we have described and analyzed the problem for the general case of *asynchronous communication*. Two conceptual mechanisms were found to be useful in understanding and presenting solutions for the problem: The *atomic model* as an abstract model of distributed computation and *time diagrams* as a convenient means of representing the overall communication scheme. The atomic model and time diagrams are also valuable for demonstrating various properties of the methods and in proving the algorithms to be correct.

A distributed system is defined as being terminated if all messages sent, are also received. This leads to the idea of *counting messages*, and in fact all our methods are based on message counting. We showed that a crude method of counting leads to false results, whereas more sophisticated counting principles can cope with inconsistent time cuts

and the lack of global time. A crucial observation is that, in the time diagram, basic messages crossing the line representing the control wave, especially those crossing over to its left side, should not be allowed to go unnoticed.

We presented several termination detection algorithms with different characteristics, and many more variants can be devised. An ideal method should cause the smallest possible overhead, allow fast detection of termination, impose no restrictions on the computation proper and be easily implementable. Other desirable properties are *symmetry*, i.e., the algorithm is syntactically identical for each process and any process may start the detection algorithm (Tan and van Leeuwen 1986; Mattern 1986) and *genericity* (processes do not have global knowledge, such as the total number of processes) (Bougé 1985a). Obviously, there is no one general method whose features are best in every case. According to the intended use and by taking into consideration the characteristcs of the underlying application and system, a method with the appropriate properties must be chosen.

Among others, the following considerations should be taken into account:

- When will the termination test be started? In most cases when the system is already terminated or near the start of the computation? (E.g., a process might start the test only when its local state indicates the possibility of global termination, or the algorithm might be triggered by a sufficiently large timeout).
- Should it be possible for an arbitrary process to initiate the test or is there a predesignated process for doing this? Is the method to cope with several concurrently active detection waves?
- Is it intended that the test should terminate as quickly as possible and possibly report a negative result, or should it remain active until the global termination state has been reached?
- Is it possible to assign a higher priority to basic messages than control messages, so that they are given preferential treatment?
- Are there any application or system dependent reasons (e.g., fixed communication channels, communication bottlenecks, one way communication etc.) for preferring one control configuration to another?
- How does the length of the control messages affect efficiency? Should their total number be minimized and what degree of concurrency is best?
- Is it feasible to alter basic messages, e.g., augment them with a time stamp?

To assist in choosing the most appropriate termination detection scheme for a particular application, we summarize and compare the main properties of our methods:

(1) The four counter method:
a) Each process must be visited at least twice.
b) The solution is symmetric and "reentrant". The processes are not aware of their identification number and local variables are not changed by the control wave. Thus an arbitrary process may start the algorithm and several control waves may be active in parallel. (In this case the control wave must carry with it a counter or the identification tag of the initiator). The algorithm is easily adapted for dynamic systems because the processes need not be aware of their total number.
c) Various control configurations are possible, for instance stars, rings or trees. Rings are symmetric but sequential, whereas stars and trees allow parallel execution of the algorithm. In the star topology a predesignated process starts the termination test. The same is true for fixed trees. If an echo implementation is used and it is possible for an arbitrary process to start the algorithm, more control information is necessary, particularly if several control waves are concurrently active.
d) There is no upper bound on the number of control messages. It is necessary to guess when the algorithm should be restarted after an unsuccessful trial.

(2) The sceptic algorithm:
a) The method is very similar to the four counter method and comments *a*, *c*, and *d* also apply here.
b) The use of flags minimizes the amount of control information (i.e., using one single accumulated counter, and in some places flags instead of counters), but in the case of concurrent activations it slightly complicates the algorithm.

(3) The time algorithm:
a) It may be sufficient to visit each node only once.
b) Symmetric and reentrant variants of the algorithm can be devised, although they are slightly more complicated than comparable implementations of the four counter method.
c) The comments on control configurations and the total number of control messages involved in the sceptic and four counter methods also apply here.
d) Each basic message must be augmented with a time stamp.

(4) The vector counter method:
a) As in the time algorithm it may be sufficient for each node to be visited just once.

b) We presented a single initiator implementation of the method on a ring where a process may not communicate with itself. However the main idea of the method is also applicable to more general cases.
c) In contrast to the time algorithm basic messages need not be changed.
d) The amount of information transferred by control messages is larger than in all the other methods.
e) Processes must be aware of their total number or at least of the number of neighboring processes to which they send basic messages.
f) In the presented implementation the test remains active until the computation proper finishes. The algorithm never reports a negative result.
g) Control messages wait for basic messages. The worst case communication complexity for the shown implementation is $O(mn)$, where $m$ denotes the number of basic messages and $n$ the number of processes.

(5) Channel counting:
a) Comments *a*, *c*, *e*, and *f* of the vector counting method also apply to this algorithm.
b) The only reasonable control configuration seems to be the scheme induced by the echo algorithm. In this implementation control messages only carry a single counter with them.
c) As in the vector counter method, control messages can wait for basic messages. The worst case communication complexity is $O(m)$. However the algorithms should not be assessed on this criterion. The exact number of control messages of both methods is highly dependent on the communication patterns of the underlying computation proper, the characteristics of the communication system and the time at which the algorithm is started.
d) In contrast to the ring implementation of the vector counter method, the echo channel counting algorithm permits concurrent execution.

The particular computational model we have used, comprises the main difference between our methods and other known solutions of the distributed termination problem. Few other methods are based on or suited to a communication oriented termination model in which message passing is asynchronous without FIFO characteristics. Termination detection is easier for *synchronous* message passing models especially since it is not necessary for the messages to be counted. Although our algorithms also work in the synchronous case, simpler variants of the sceptic algorithm (Francez and Rodeh 1982; Dijkstra et al. 1983; Topor 1984) or other synchronous solutions of the problem (Arora and Sharma 1983; Szymanski et al. 1985) should also be taken into consideration.

If message passing is not instantaneous, but upper bounds exist on message delay, solutions with time-out mechanisms are conceivable (Lozinskii 1984, 1985). In systems where messages obey the *FIFO rule*, another principle can be applied: a control message can clear the communication channels by sweeping all basic messages before it. Misra makes use of this principle but although he presents a distributed algorithm his solution is inherently sequential; a single marker traverses all edges of the communication graph (Misra 1983). A better solution in which markers are sent out in parallel, is presented in a recent paper on a quiescent properties detection paradigm (Chandy and Misra 1985). If control messages have a lower priority than basic messages, solutions with bounded memory and $O(m)$ message complexity are conceivable (Rozoy 1986).

Our algorithms are not intended for *dynamic systems*, in which processes can be dynamically generated. Cohen and Lehmann discuss solutions for such systems which are based on "responsibility trees" (Cohen and Lehmann 1982). In their model a process which creates a new process becomes responsible for it, but destruction of processes is not possible. Lai also gives solutions to the distributed termination problem for dynamic systems (Lai 1985, 1986). Our methods, particularly those where the processes are not aware of their total number, can be extended to cope with dynamic systems: A process creating a new one sends a virtual basic message to the newly generated process, to itself, or to some other dedicated process (e.g. the initiator). The message is only considered as being received when the new process has been integrated into the control topology.

How realistic is the atomic model of distributed computation? The *normal form* of CSP programs (e.g., Apt 1986) where I/O commands are only allowed in guards at an outer main loop, is actually a transactional programming style. However in general, it is not necessary to impose such a restrictive structure. In systems based on models which are not transaction oriented (i.e., implicitly received messages trigger the execution of uninterruptible operations), but where messages can be received at arbitrary places (e.g., programming languages with an explicit receive statement), the programmer has to ensure that whenever no more messages are being sent, the local termination state is signalled at appropriate parts of the program. In order to avoid blocking processes in an unnoticed deadlock

state, a process should accept control messages whenever it is waiting for basic messages.

It should be easy to adapt the proposed termination detection methods to characteristics of various systems and applications. Most of the algorithms have been realized in the distributed programming language CSSA (Mattern and Beilken 1985) implemented on the INCAS experimental distributed system (Nehmer et al. 1987). The principles should also be applicable to other quiescence and persistent global state detection problems, e.g. distributed infimum approximation (Tel 1986) or concurrent on-the-fly garbage collection (Tel et al. 1986) and problems from related areas such as distributed debugging or distributed databases.

*Acknowledgments.* The author would like to thank Christian Beilken, Mike Reinfrank and Mike Spenke, for valuable comments and many fruitful discussions, and Jackie Randell, who revised the English text. The assistance of the other members of the SFB124 research project is also gratefully acknowledged. Special thanks to the referees for their comments.

## References

Apt KR, Richier J-L (1985) Real time clocks versus virtual clocks. In: Broy M (ed) Control flow and data flow: Concepts of distributed programming. Springer, Berlin Heidelberg New York, pp 475–501

Apt KR (1986) Correctness proofs of distributed termination algorithms. ACM Trans Program Lang Syst 8(3):388–405

Arora RK, Sharma NK (1983) A methodology to solve distributed termination problem. Inf Syst 8(1):37–39

Beilken C, Mattern F, Reinfrank M (1985) Verteilte Terminierung – ein wesentlicher Aspekt der Kontrolle in verteilten Systemen. Report SFB124-41/85, Department of Computer Science, University of Kaiserslautern, FRG

Bougé L (1985a) Symmetry and genericity for CSP in distributed systems. Report 85-32, LITP, Universite Paris 7, France

Bougé L (1985b) Repeated synchronous snapshots and their implementation in CSP. In: Brauer W (ed) 12th Coll Automata, Lang and Programming. Springer, Berlin Heidelberg New York, LNCS 194, pp 63–70

Chandy KM, Misra J, Haas LM (1983) Distributed deadlock detection. ACM Trans Comput Syst 1(2):144–156

Chandy KM, Lamport L (1985) Distributed snapshots: Determining global states of distributed systems. ACM Trans Comput Syst 3(1):63–75

Chandy KM, Misra J (1985) A paradigm for detecting quiescent properties in distributed computations. In: Apt KR (ed) Logics and models of concurrent systems. Springer, Berlin Heidelberg New York, pp 325–341

Chandy KM, Misra J (1986) An example of stepwise refinement of distributed programs: quiescence detection. ACM Trans Program Lang Syst 8(3):326–343

Chang EJH (1982) Echo algorithms: Depth parallel operations on general graphs. IEEE Trans Software Eng SE-8(4):391–401

Clinger WD (1981) Foundations of actor semantics. Report AI-TR-633, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, USA

Cohen S, Lehmann D (1982) Dynamic systems and their distributed termination. Proc ACM SIGACT-SIGOPS Symp Principles Distributed Comput, Ottawa, pp 29–33

Dijkstra EW, Scholten CS (1980) Termination detection for diffusing computations. Inf Process Lett 11(1):1–4

Dijkstra EW, Feijen WHJ, van Gasteren AJM (1983) Derivation of a termination detection algorithm for distributed computations. Inf Process Lett 16(5):217–219

Francez N (1980) Distributed termination. ACM Trans Program Lang Syst 2(1):42–55

Francez N, Rodeh M, Sintzoff M (1981) Distributed termination with interval assertions. In: Diaz J, Ramos I (eds) Proc Int Colloq Formalization of Programming Concepts. Springer, Berlin Heidelberg New York, LNCS 107, pp 280–289

Francez N, Rodeh M (1982) Achieving distributed termination without freezing. IEEE Trans Software Eng SE-8(3):287–292

Kumar D (1985) A class of termination detection algorithms for distributed computations. In: Maheshwari N (ed) 5th Conf on Foundations of Software Technology and Theoretical Computer Science, New Delhi. Springer, Berlin Heidelberg New York, LNCS 206, pp 73–100

Lai T-H (1985) Termination detection for dynamic distributed systems with non-first-in-first-out communication. Report 85-16, Computer and Information Science Research Center. Ohio State University, Columbus, USA

Lai T-H (1986) A termination detector for static and dynamic distributed systems with asynchronous non-first-in-first-out communication. In: Kott L (ed) 13th Coll Automata, Lang and Programming. Springer, Berlin Heidelberg New York, LNCS 226, pp 196–205

Lamport L (1978) Time, clocks and the ordering of events in a distributed system. Commun ACM 21(7):558–565

Lavallee I, Roucairol G (1986) A fully distributed (minimal) spanning tree algorithm. Inf Process Lett 23:55–62

Lozinskii EL (1984) Yet another distributed termination. Report 84-2, Department of Computer Science, The Hebrew University of Jerusalem, Israel

Lozinskii EL (1985) A remark on distributed termination. Proc 5th Internat Conf on Distributed Computing Systems, Denver, pp 416–419

Mattern F, Beilken C (1985) The distributed programming language CSSA – a short introduction. Report 123/85, Department of Computer Science, University of Kaiserslautern, FRG

Mattern F (1986) Asynchronous distributed termination – parallel and symmetric solutions with echo algorithms. Report SFB124-21/87, Department of Computer Science, University of Kaiserslautern, FRG

Misra J, Chandy KM (1982) Termination detection of diffusing computations in communicating sequential processes. ACM Trans Program Lang Syst 4(1):37–43

Misra J (1983) Detecting termination of distributed computations using markers. Proc 2nd Ann ACM Symp Principles Distributed Comput, Montreal, Quebec, pp 290–294

Nehmer J et al. (1987) Key concepts of the INCAS multicomputer project. IEEE Trans Software Eng SE-13(8):913–923

Rana SP (1983) A distributed solution of the distributed termination problem. Inf Process Lett 17(1):43–46

Richier J-L (1985) Distributed termination in CSP: Symmetric solutions with minimal storage. In: Mehlhorn K (ed) Proc STACS 85. Springer, Berlin Heidelberg New York, LNCS 182, pp 267–278

Rozoy B (1986) Model and complexity of termination for distributed computations. In: Gruska J, Rovan B, Wiedermann

J (eds) Proc Math Found Comp Sc 86. Springer, Berlin Heidelberg New York, LNCS 233, pp 564–572
Shavit N, Francez N (1986) A new approach to detection of locally indicative stability. Report RC 11925 (# 53703), IBM Thomas J Watson Research Center, Yorktown Heights, USA
Tan RB, van Leeuwen J (1986) General symmetric distributed termination detection. Report RUU-CS-86-2, Department of Computer Science, University of Utrecht, The Netherlands
Tel G (1986) Distributed infimum approximation. Report RUU-CS-86-12, Department of Computer Science, University of Utrecht, The Netherlands
Tel G, Tan RB, van Leeuwen J (1986) The derivation of graph marking algorithms from distributed termination detection protocols. Report RUU CS-86-11, Department of Computer Science, University of Utrecht, The Netherlands
Topor RW (1984) Termination detection for distributed computations. Inf Process Lett 18(1):33–36
Szymanski B, Shi Y, Prywes NS (1985) Synchronized distributed termination. IEEE Trans Software Eng SE-11(10):1136–1140

## Note added in proof

Since the first version of this article was submitted, many papers on the distributed termination problem have been published. The following list completes the references:

Afek Y, Saks M (1987) Detecting global termination conditions in the face of uncertainty. Techn Rep, Bell Laboratories, Murray Hill, USA
Augusteijn L (1986) Establishing global assertions in a distributed environment. ESPRIT project 415, Doc. No. 183, Philips Research Laboratories, Eindhoven, The Netherlands
Arora RK, Rana SP, Gupta MN (1986) Distributed termination detection algorithm for distributed computations. Inf Process Lett 22:311–314 [The algorithm is wrong. See Tan RB, Tel G, Van Leeuwen J (1986) Comments on "Distributed termination detection algorithm for distributed computations". Inf Process Lett 23:163
Chandrasekaran S, Kannan CS, Venkatesan S (1987) Efficient distributed termination detection. Proc IFIP Conf Distributed Processing. North-Holland, Amsterdam New York Oxford
Dijkstra EW (1987) Shmuel Safra's version of termination detection. Report EWD998-0, Department of Computer Science, University of Texas at Austin, USA
Erikson O, Skyum S (1986) Symmetric distributed termination. In: Rozenberg G, Salomaa A (eds) The book of L. Springer, Berlin Heidelberg New York, pp 427–430
Ferment D (1986) Finite or not finite solutions for the distributed termination problem. Report 86-11, LITP, Universite Paris 7, France
Ferment D, Rozoy B (1986) Possibility and impossibility of solutions for the distributed termination problem. Report 86-8, LITP, Universite Paris 7, France
Ferment D, Rozoy B (1987) Solutions for the distributed termination problem. In: Albrecht A, Jung H, Mehlhorn K (eds) Parallel algorithms and architectures. Springer, Berlin Heidelberg New York, LNCS 269
Hazari C, Zedan H (1987) A distributed algorithm for distributed termination. Inf Process Lett 24:293–297 [It is not the 'classical' distributed termination problem: processes are never reactivated. See also Tel G, Van Leeuwen J (1987) Comments on "A distributed algorithm for distributed termination". Inf Process Lett 25:349]
Helary J-M, Jard C, Plouzeau N, Raynal M (1987) Detection of stable properties in distributed applications. Proc 6th ACM Symp Principles Distributed Comput
Koo R, Toueg S (1987) Effects of message loss on distributed termination. Tech Rep, Department of Computer Science, Cornell University, USA
Lai T-H (1986) Termination detection for dynamic distributed systems with non-first-in-first-out communication. J Parallel and Distributed Computing 3:577–599
Lai T-H (1987) Message-optimal algorithms for termination detection in broadcast networks. Department of Computer and Information Science, The Ohio State University, USA
Mattern F (1987) Experience with a new distributed termination detection algorithm. In: Gafni E, Raynal M, Santoro N, Van Leeuwen J, Zaks S (eds) Proc 2nd Int Workshop Distributed Algorithms, Amsterdam. Springer, Berlin Heidelberg New York, LNCS
Mattern F (1987) An efficient distributed termination test. Report SFB124-32/87, Department of Computer Science, University of Kaiserslautern, FRG
Müller H (1987) High level petri nets and distributed termination. In: Voss K, Genrich HJ, Rozenberg G (eds) Concurrency and nets. Springer, Berlin Heidelberg New York, pp 349–362
Roucairol G (1987) On the construction of distributed programs. In: Paker Y, Banatre J-P, Bozyigit M (eds) Distributed operating systems: theory and practice. Springer, Berlin Heidelberg New York, pp 47–65
Saikkonen H, Ronn S (1986) Distributed termination on a ring. BIT 26:188–194
Sanders BA (1987) A method for the construction of probe-based termination detection algorithms. Proc IFIP Conf Distributed Processing. North-Holland, Amsterdam New York Oxford
Shavit N, Francez N (1986) A new approach to detection of locally indicative stability. In: Kott L (ed): 13th Coll Automata, Lang and Programming. Springer, Berlin Heidelberg New York, LNCS 226, pp 344–358
Verjus JP (1987) On the proof of a distributed algorithm. Inf Process Lett 25:145–147
Zöbel D (1986) Programmtransformationen zur Ende-Erkennung bei verteilten Berechnungen. Informationstechnik 28(4):204–213